

AD-A222 883

A Computer for Low Context-Switch Time

TR90-012

March, 1990

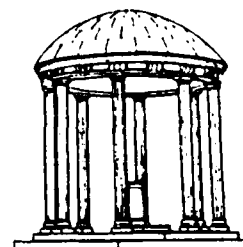


Mark Charles Davis

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175



This research was supported by the Office of Naval Research, Contract N00014-86-K-0680.

UNC is an Equal Opportunity/Affirmative Action Institution.

90 05 29 173

A Computer for Low Context-Switch Time

by
Mark Charles Davis

A dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1990


Approved by:



Advisor: Frederick P. Brooks, Jr.



Reader: Donald F. Stanat


Reader: Kye S. Hedlund

© 1990
Mark Charles Davis
ALL RIGHTS RESERVED

MARK CHARLES DAVIS. A Computer for Low Context-Switch Time (Under the direction of Frederick P. Brooks, Jr.)

ABSTRACT

A context switch is the suspension of one running process and the activation of another in a multitasking environment. Many applications, such as process control, require frequent context switches among many processes. A context switch requires a substantial amount of time: about 1000 microseconds on a VAX 11/780 and about 500 microseconds on Sun 4/280. Recently introduced computer architectures, such as the Sun 4, have not improved context-switch performance as much as they have improved throughput. A computer architecture with appropriate memory hierarchy can give better support to context switching. The Computer for Low Context-Switch Time (CLOCS) is a computer with such an architecture. Because the architecture has minimum state inside the Central Processing Unit, CLOCS can switch context in less than the time required to execute one instruction. The CLOCS Memory Management Unit provides virtual memory without degrading context-switch time as long as the new process is located in physical memory. Analyses of the architecture show that CLOCS throughput performance approaches the performance of contemporary RISC workstations and that it is well suited for real-time applications. Because these analyses showed promise for the CLOCS architecture, a register-transfer level implementation was designed and simulated to estimate more accurately the performance of a feasible CLOCS computer system. Although many standard implementation techniques were not useful, a technique called short-circuiting reduced memory references by 15%. On the Dhrystone integer benchmark program, CLOCS performed at least 30% as fast as contemporary workstations constructed from the same electronics technologies, and further significant improvement of CLOCS performance is possible. By using this lower bound on CLOCS throughput performance, the proper architecture can be identified for an application with challenging context-switch requirements.

STATEMENT "A" per Dr. A. Van Tilborg
ONR/Code 1133
TELECON

6/11/90

VG



By <i>pes call</i>	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
<i>A-1</i>	

ACKNOWLEDGMENTS

As with any significant research, this work would not have been possible without the help of many others. I would like to acknowledge their contributions here.

First, I would like to thank my advisor for his inspiration and guidance. Without Fred Brooks' perception of important topics and encouragement, I never would have discovered anything. The other members of my committee, Kye Hedlund, Don Stanat, Dean Brock, and Akhilesh Tyagi, made great contributions. Also notable was Jay Nievergelt; while serving on my committee, he forced me to broaden my horizons, and as I responded to his challenges, my study gained wider applicability.

I was also helped by many other people at the University of North Carolina. Bill Gallmeister worked with me, helped me design the CLOCS architecture and taught me quite a bit about UNIX. Several other students participated in class projects that produced software valuable to our research effort. Michael Winslett, Curtis Hill and Suresh Rajgopal produced the first CLOCS C language compiler. David Cox and Matt Fitzgibbon wrote a prototype operating-system kernel that Bill used to estimate CLOCS context-switch performance. David Lines helped me whip this document into shape.

Several organizations have greatly helped my efforts. The Office of Naval Research has supported CLOCS researchers and provided the computer equipment and supplies we needed through Contract N00014-86-K-0680. The Computer Science Department of the University of North Carolina has also provided valuable facilities. The Microelectronics Center of North Carolina provided the consulting services of Fred Heaton, an experienced microprocessor designer, to keep our feet on the ground. The Free Software Foundation of Cambridge, Massachusetts, provided the GNU C language compiler and many other high-quality software tools.

Of course, my family has been quite indulgent and supportive. But more than that, my wife, Diane Holditch-Davis, has given me invaluable advice on research methodology and statistics.

TABLE OF CONTENTS

List of Figures	x
List of Tables	xi
List of Abbreviations	xii
I. Introduction	1
1.1 Context-Switch Time May Be Reduced by Architecture	1
1.2 Context Switching	2
1.2.1 Context Switching on Existing Computer Systems	2
1.3 Why Does Context Switching Take So Long?	4
1.4 Why Is Context Switching Important?	5
1.5 The Impact of Memory Hierarchy on Context Switching	6
1.6 How CLOCS Meets the Thesis Goals	8
1.7 The Tradeoff: Throughput vs. Context Switching	9
1.8 Summary and Layout of Research	11
II. Previous Work	12
2.1 IBM 801	12
2.2 Berkeley RISC	13
2.3 AT&T CRISP	14
2.4 MIPS Company R2000	15
2.5 Sun Microsystems SPARC	15
2.6 Other New Architectures	16
2.7 Texas Instruments 9900	17
2.8 Atlas	17
2.9 Conclusions	18
III. Design Considerations Other Than Context Switching	19
3.1 Features Needed to Support General Purpose Computing	20
3.1.1 Adequate Address Space	20
3.1.2 Adequate Calculation Precision	21
3.1.3 Adequate Variety of Data-Object Sizes	21

3.1.4	Expensive Operations to Be Cheaply Emulated	22
3.1.5	Predictable Worst Case Performance	23
3.2	Operating System Support Features	23
3.2.1	Virtual-Memory Support	23
3.2.2	Shared-Memory Support	24
3.2.3	Classes of Main-Storage Areas	24
3.2.4	Semaphores	24
3.3	Compiler Support Features	24
3.3.1	Data-Object Size	25
3.3.2	Addressing Mode Expectations	25
3.4	Compiler Properties Required to Support the Architecture	26
3.4.1	Compiler Optimization	26
3.4.2	Support Different Data-Object Sizes	26
IV.	Description of the Architecture	27
4.1	Highlights	27
4.1.1	Noteworthy	27
4.1.2	Peculiarities	28
4.1.3	History	28
4.2	Instruction Formats	28
4.2.1	Address Specification	29
4.2.2	Variations of Field Use	29
4.2.3	Spaces and Addressing	30
4.3	Programming Model	30
4.3.1	Control Storage	30
4.3.2	Address Calculation and Addressing Modes	32
4.3.3	Addressing Mode Summary	35
4.4	Data Formats	35
4.4.1	Fixed Point	35
4.4.2	Floating Point	35
4.4.3	Character	36
4.5	Operations	36
4.5.1	Decision	37
4.5.2	Data Operations	37
4.5.3	Sequencing	38
4.5.4	Supervisory	39
4.5.5	Input and Output	40
4.6	Implementation Notes	40
V.	Description of the Memory Management Unit	41
5.1	Organization of the Memory Management Unit	41
5.1.1	Requirements	41
5.1.2	What the MMU Does	41
5.1.3	External Appearance of the MMU	43

5.1.4	Physical Page Status	44
5.2	Contents of the MMU Word	45
5.2.1	Sizing Considerations for Fields	45
5.2.2	Mapping of Word Use to <i>FLG</i>	48
5.3	MMU Operations and Exceptions	49
5.3.1	Normal Read and Write	49
5.3.2	From <i>PID, VP</i> Get <i>PP</i> for an Operand and Check Permissions	50
5.3.3	From <i>PID, VP</i> Get <i>PP</i> for Text and Check Permissions	50
5.3.4	From <i>PID, OSID, VP</i> Get <i>PP</i> , and Check Permissions	50
5.3.5	From <i>PID, ISID, VP</i> Get <i>PP</i> , and Check Permissions	50
5.3.6	Change Primary <i>OSID</i>	51
5.3.7	Change Primary <i>ISID</i>	51
5.3.8	Reset <i>USED</i> for All Physical Pages	52
5.4	Implementing Common Virtual-Memory Operations	52
5.4.1	Write-Back Virtual Memory	52
5.4.2	Copy-on-Write	53
5.4.3	Not-Used-Recently Page Replacement	53
5.5	The MMU Designs We Discarded	53
5.5.1	Alternate A - Virtual and Physical Tables	53
5.5.2	Alternate B - Some Registers Permanently Mapped	55
VI.	Quantitative Analysis of CLOCS	57
6.1	Programming a Small Problem on CLOCS	57
6.2	Expected Memory-Bandwidth Requirements	59
6.3	Favorable Comparison with MIPS R2000	61
6.4	Using the "Measurement and Evaluation of Alternative Computer Architectures" Technique	62
6.4.1	Fuller's Criteria	63
6.4.2	CLOCS and Other Architectures	65
6.5	Summary	65
VII.	Implementation of the CLOCS Architecture	68
7.1	Why Implement CLOCS?	68
7.2	What Does an Implementation Look Like?	70
7.3	What Implementation Techniques May Be Used?	71
7.3.1	Pipelining	71
7.3.2	Data Caching	72
7.3.3	Multiported Memory	73
7.3.4	Posted Write	73
7.4	Chosen Implementation	74
7.4.1	Implementation Overview	74
7.4.2	Practicality of this Implementation	77
7.4.3	Performance Assessment	77
7.4.4	Implementation Feature Additions	79

7.5	Discarded Designs	80
7.5.1	Functional	80
7.5.2	Seven-Instruction Pipeline	81
7.5.3	Two-Instruction Pipeline Overview	82
7.5.4	Simulation Results and Conclusions	85
VIII.	Simulation Programs and Results	86
8.1	Simulation System Description	86
8.1.1	GNU C Compiler	86
8.1.2	Assembler	88
8.1.3	Implementation-Level Simulator	88
8.1.4	Scaffolding	89
8.1.5	Library Routines	90
8.2	Benchmark Programs	91
8.2.1	Short Test Programs	91
8.2.2	Dhrystone 2.1	91
8.3	Simulation Results	91
8.3.1	Instruction Mix	91
8.3.2	Effect of Memory System Design	92
8.3.3	Effect of GCC Optimizations	94
8.4	Findings: When CLOCS Pays	94
IX.	Conclusions and Future Work	98
9.1	CLOCS Potential Performance Is Close to Contemporary RISC	98
9.2	CLOCS Fuller-Type Analysis Score Is High	98
9.3	CLOCS Uses 3.2 Memory References per Instruction	99
9.4	Short-Circuiting Saves 15% of Memory References	99
9.5	Feasible CLOCS System Performance Is Good	99
9.6	When Is a New Architecture Indicated?	99
9.7	When a Conventional Architecture Is Indicated	100
9.8	Register File Sharing	100
9.8.1	Register Backing Store	101
9.8.2	Register Trickle Refill	101
9.9	Programming Language Observations	102
9.9.1	Dynamic Allocation Inefficient	103
9.9.2	Recursion Works Poorly	103
9.10	Future Work	103
9.11	Improve the C Language Compiler	104
9.11.1	Improve the CLOCS Architecture for the C Language	105
9.11.2	Try Another Language: FORTRAN	105
	Bibliography	105

A.	Detailed Simulation Results	110
A.1	General Performance	110
A.2	Various Memory Designs	110
A.3	Effect of GCC Optimizations	110
B.	Source Listings	116
B.1	Arrangement of Support Software	116
B.1.1	CCC (C Language Compiler for CLOCS)	116
B.1.2	GCCC (GNU C Language Compiler for CLOCS)	117
B.1.3	CASM (CLOCS Assembler)	117
B.1.4	CLOADER (CLOCS Loader) and CDUMP (CLOCS Ob- ject Module Dumper)	117
B.1.5	CAS	117
B.2	Arrangement of Application Software	118
B.2.1	C Library Routines	118
B.2.2	Test Programs	124
B.2.3	Dhrystones Version 2.1	128

LIST OF FIGURES

1.1	Memory Hierarchies	6
1.2	Comparison of CLOCS and Conventional Architectures	10
4.1	CLOCS Instruction Format	28
4.2	CLOCS Status Word Format	31
5.1	What the CLOCS MMU Does	42
5.2	CLOCS MMU Address Translation (Detailing of Figure 5.1)	46
5.3	CLOCS MMU Word Format	46
6.1	Solution to Exercise J-2 from Blaauw and Brooks	58
7.1	CLOCS Implementation Overview	74
7.2	Operand and Alu Portion of Chosen Implementation	76
7.3	Memory and Check Portion of Chosen Implementation	78
7.4	Two-Instruction Pipeline Implementation	84
8.1	Comparison of CLOCS and a DECStation 3100	96
9.1	Comparison of Improved CLOCS and a DECStation 3100	106
B.1	Strcmp.c Source	119
B.2	Strcmp.s Source	120
B.3	Strcpy.c Source	121
B.4	Strcpy.s Source	121
B.5	Malloc.s Source	122
B.6	Scanf.c Source	122
B.7	Scanf.s Source	123
B.8	Printf.s Source	123
B.9	Times.s Source	124
B.10	Assign.c Source	124
B.11	Lloop.c Source	125
B.12	Sub.c Source	125
B.13	Quicksort.c Source (Subroutines)	126
B.14	Quicksort.c Source (Main Program)	127

LIST OF TABLES

1.1	Measured Context-Switch Time	3
1.2	Representative Processes and Context-Switch Rates	4
4.1	Summary of Addressing Modes	36
5.1	All Possible MMU Page Conditions	48
5.2	MMU Condition Assignments	49
5.3	Comparison of Chosen and Alternate A MMU Designs	55
6.1	Comparison of CLOCS and R2000 Memory Use	61
6.2	Results of a Fuller-Type Analysis	66
8.1	Percentage of Operations for Dhrystone Benchmark Program	92
8.2	Effect of Short Circuits on Memory Access	93
8.3	Memory-Design Effect on Dhrystone Performance	93
A.1	General Performance Results of CLOCS	111
A.2	Cycle Counts for Various Programs and Memory Systems	112
A.3	Memory Design Effect on Dhrystone Cycles per Instruction	113
A.4	GCC Optimizations for Dhrystones	114
A.5	Effect of GCC Optimizations (Percentages)	115

LIST OF ABBREVIATIONS

ALU	arithmetic/logic unit
CAS	CLOCS architecture simulator
CASM	CLOCS assembler
CISC	comprehensive instruction set computer
CLOCS	the Computer for low context-switch time
CPU	central processing unit
DEC	Digital Equipment Company, Inc.
GCC	the GNU C compiler
ISID	instruction segment identifier
MMU	memory management unit
OSID	operand segment identifier
PID	process identifier
RISC	reduced instruction set computer
SID	segment identifier
TI	Texas Instruments, Inc.
VLSI	very large scale integration

Chapter I

Introduction

The computer for low context-switch time, CLOCS, is a computer architecture designed to reduce context-switch time by using an unconventional memory hierarchy. This chapter presents the CLOCS thesis. Then sections define context switching and detail relevant applications. The chapter concludes with an interpretation of the thesis and a description of the resulting architectural features.

1.1 Context-Switch Time May Be Reduced by Architecture

Many computer applications require rapid switching between independent tasks; however, most recent computer architecture research has emphasized throughput more than context-switch performance. As a result, the new computer designs have worse relative context-switch performance: the time needed to switch contexts has not decreased as much as the time to run programs. Much of the relative performance loss has come as larger register sets are added to the Central Processing Units (CPU). Other features of architectures and implementations (for example, caches) improve throughput at the expense of context-switch time. My thesis runs contrary to this trend:

For applications with sufficiently high occurrence of context switching among large numbers of tasks, best performance may be attained from a computer with fewer, broader levels of memory hierarchy.

Two developments in the last few years have increased the credibility of this thesis:

- Real-time operating systems and process-control applications have increased the frequency of context switching. Faster computers can support more complicated applications; this results in more context switches. Even general pur-

pose operating systems such as UNIX have much higher frequency of context switching than is commonly assumed.

- Advances in Very Large Scale Integrated Circuit (VLSI) technology have reduced the speed differential between different sizes of memory. For example, an eight kilobyte on-chip cache may be accessed almost as rapidly as a 64 byte register file.

Section 1.6 details the architecture features suggested by the thesis, but the most important characteristics are that such a computer has no registers and has no cache.

The main question of this study is: How seriously is throughput performance impaired by this approach to computer architecture? Reasonable estimates are possible only through the detailed design of a computer system, including support software and implementation specification. But, first, let us examine context switching in more detail.

1.2 Context Switching

A context switch is the suspension of a running program and the activation of another. It differs from a subroutine call in that the running program does not know specifics about the task to be activated next, and the new program may have different ownership or other characteristics.

In many operating systems, system service requests use a trap instruction that effectively changes the context. The ownership and permission of the operating system are assumed as the trap occurs. Sometimes this type of context switch is called a *system call* and may be less expensive than activating another separate task. For example, UNIX minimizes the expense of system calls by the having the kernel share the virtual address space with all application programs, so no adjustments to the virtual memory system are required. Although the CLOCS operating system does system calls by full context switching, when I refer to context switches on existing computers, I will exclude system calls.

1.2.1 Context Switching on Existing Computer Systems

Switching context requires hundreds of microseconds. Lefler [27] reports that a Digital Equipment Company (DEC) VAX 11/780 running 4.2BSD UNIX requires 280

Computer Type	Context Switch Time (in milliseconds)
VAX 11/780	1.0
VAX 11/750	2.0
Sun 2/50	1.7 - 6.0
Sun 3/75	0.80 - 2.5
Sun 4/280	0.5 - 1.9
DECStation 3100	0.25

Note: The range of context-switch times for the Sun computers represents the range of performance when two (shorter time to switch) to 32 (longer time) processes are active[8].

Table 1.1: Measured Context-Switch Time

microseconds for a system call and 4.4 milliseconds for a context switch. Feder [14] reports context-switch times for the 780 of 700 microseconds for UNIX 4.0 and of 400 microseconds when running system V on a VAX 780. He also reports context-switch times of about 500 microseconds for a 3b20S. Our measurements of context-switch time confirm these figures.

In more recent measurements done at the University of North Carolina[8], a Sun 4 context switch required about 500 microseconds. Less than 100 microseconds were required to save the hardware state; the remainder was consumed by scheduling and related activities of the operating system. However, the Sun 4 context-switch time may increase to over 1250 microseconds due to required adjustments to the memory management system if more than 16 processes are frequently active. This computer has room for 16 different virtual memory contexts. When a process is activated that does use one of the stored contexts, activation takes much longer. If the longer activations occur frequently, the average context-switch time becomes longer. If each newly activated process requires a new memory management context, then an extra 750 microseconds are added to each context switch, making the average time 1250 microseconds. The increased speed of the Sun 4 barely compensates for the much larger amount of CPU state (represented in part by 192 registers) that must be saved on a context switch. The context-switch time for several types of computers at the University of North Carolina Computer Science Department is shown in Table 1.1. Table 1.2 contains representative values for the number of processes and context switches per second of the department's computers.

Computer Type	Number of Processes	Switches per Sec
Shared VAX	250	40-100
Utility VAX	50	35
Workstation	25-50	50 - 600
File Server	35	100 - 500

Table 1.2: Representative Processes and Context-Switch Rates

1.3 Why Does Context Switching Take So Long?

Context switching is time consuming because several things must be accomplished:

1. Save general purpose registers.
2. Save floating-point registers and related state.
3. Save program counter and other CPU state.
4. Adjust memory management (if required).
5. Select next task to run.
6. Restore new task's state (same as saved in 1, 2, and 3).
7. After the context switch, some performance degradation may occur as cache fills, adding to the effective time required.

The time required for item 5 depends on the operating system. As mentioned above, about 80% of the time spent on a context switch goes to task scheduling. How the remainder of the time is allocated depends on the computer architecture. For example, a computer without a memory cache does not experience item 7. A computer with 192 registers will spend more time saving and restoring them than a computer with 32.

Other operating system approaches[17] may reduce the time required for scheduling during a context switch to as few as one hundred instructions (about 10 microseconds). Reducing the time required for the other items requires a new architecture based on principles different from the ones used for the VAX, the Sun 4, and other conventional architectures. To find such a better architecture is the purpose of CLOCS research.

1.4 Why Is Context Switching Important?

Although context switching is quite common, it does not significantly affect general purpose computers. At the University of North Carolina Computer Science Department, the main time-sharing computer in the worst case spends only 10% of its time on context switching. A radical new architecture to improve performance by eliminating that 10% would likely lose 10% performance elsewhere. So what types of applications need fast context switching? Here are some examples:

1. The Microelectronics System Laboratory, a computer-prototype building group at the University of North Carolina, has invested considerable programming resources in developing a UNIX compatible real-time operating system. UNIX is desirable on process-control machines to provide a reasonable program development environment.
2. The MegaOne, a state-of-the-art integrated circuit tester has two microprocessors: one runs UNIX to provide a good user interface and program development system; the other processor runs custom software to control the tester. In this case, additional hardware has been added to handle general purpose and real-time functions.
3. When the Andrew system[30] file manager, VICE, was originally designed, the file server assigned each client a separate process. Later, significant reprogramming was required because this elegant design spent too much time on context switching.
4. A six-legged walking machine has been built at Ohio State University[35] that relies on sixty processes to control the legs. The controller is constructed of a large amount of specialized computing equipment, and the software was difficult to write.

Each of these applications can benefit from a computer system that handles many independent processes and frequent switching between the processes. More aggressive applications will require even more processes and faster context-switch rates. For example, a bipod walking machine with arms will require more processes than the six legged model because of higher articulation is required for a bipod, and faster context-switch time because of the dynamic requirements to maintain balance. Such a system may require several hundred independent tasks and thousands of context

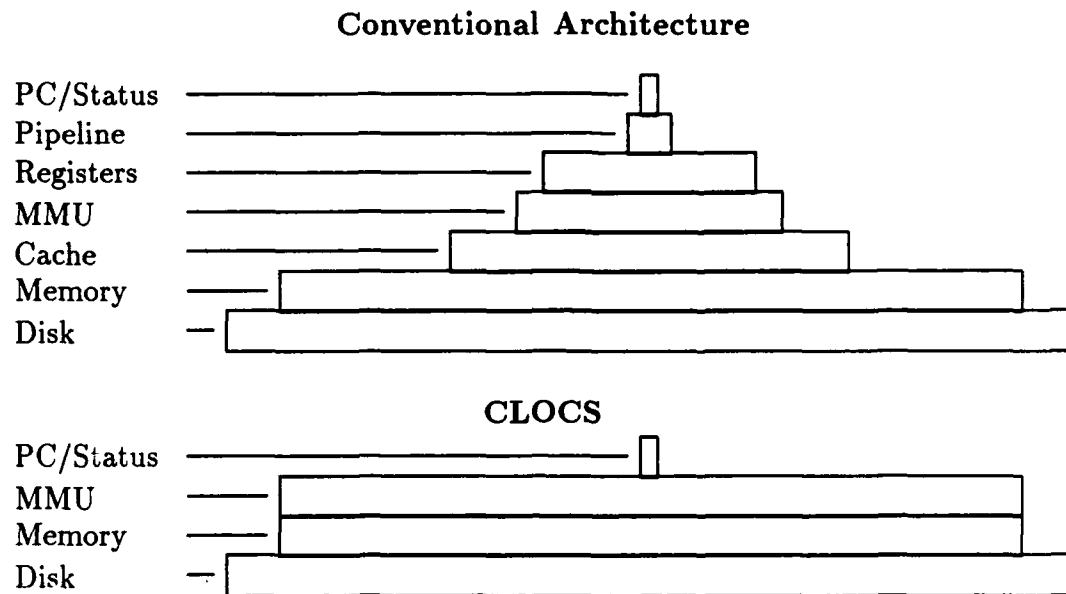


Figure 1.1: Memory Hierarchies

switches per second. Also notable, each of the applications listed above requires a general purpose operating system to support development or provide services such as protection.

Perhaps most notable are the criteria used by the Army/Navy Computer Family Architecture Committee in its study of computer architectures[16]. The criteria clearly emphasized fast context switching; several criteria valued low memory traffic for context switches and short interrupt latency. However, other criteria included features such as virtual memory, protection, and large address spaces to support sophisticated operating systems. The computers they sought had to run powerful operating systems and switch context quickly.

1.5 The Impact of Memory Hierarchy on Context Switching

Most recently designed computer architectures have several levels of memory hierarchy. Figure 1.1 shows the levels of memory hierarchy in a generic, modern computer system. In general, the items at the top of the hierarchy are smaller and faster to access than lower items.

The diagram at the top of the figure shows a memory hierarchy for a conventional

architecture. At the very highest point on the memory hierarchy is the program counter and status. Normally, this is only one or two words long and indicates the current instruction location, permissions of the running process, and status flags such as the result of the last comparison. If the computer is highly pipelined, several instructions may be held in the CPU. The third level of hierarchy is the set of registers. The Memory Management Unit (MMU) contains virtual memory addressing and protection information for one or more processes. The cache contains some of the most recently referenced instructions and data. The main memory is the next level of hierarchy and is much larger than the cache. Finally, the disk storage (which also represents the virtual memory) is the foundation of the hierarchy. This diagram describes a generic computer; actual designs will differ. For example, some architectures may not have visible instruction pipelines or support instruction continuation, so the pipeline level would not exist. Some cache memories contain only information for one process and move higher on the hierarchy than the MMU. However, these minor differences in architectural approaches do not directly bear on the context-switch discussion.

The memory hierarchy at the bottom of Figure 1.1 represents an architecture based on the CLOCS principles. Several of the levels of the hierarchy have been removed, and the MMU has expanded to be the same width as the memory.

The conventional architectures have very successfully used memory hierarchy because of the locality of reference of programs. Most programs use the same data objects repeatedly and execute the same instructions several times. This locality in time and space results in efficient use of smaller, faster memories, supported by larger, slower (and cheaper) memory systems. Thus a program can run with most of its instructions in cache and most of its data in registers or in cache. Transfers from the larger, slower main memory to the faster ones are rare, so the improved performance from using the faster memory dominates.

When context switching occurs frequently, many programs are using information instead of just one. Even though each program may have good locality of reference, the combination of all of the programs references more information than can be held by the upper levels of the hierarchy. In this case, the time to transfer information between the levels of hierarchy may dominate performance. With several programs switching context frequently, the higher levels of hierarchy impede performance, and fewer levels would give higher performance.

1.6 How CLOCS Meets the Thesis Goals

CLOCS is a computer architecture that reduces memory hierarchy. This computer architecture meets the requirements of the thesis and is otherwise simple enough for easy implementation so realistic comparisons may be made with existing machines.

The CLOCS CPU has the minimum possible state; a single word contains a program counter and other program information. To switch context, the CPU simply stores that status word and loads another. As a result, there are no other registers in the CPU, and all operations must have operands in memory. A large instruction size allows addresses to be fully specified in the instruction.

In order to support a general purpose operating system, a MMU is required. The guiding principle for the MMU is that if information (data or program) is in main storage, the MMU must support access to it without delay. This means that the MMU must be able to address any memory location, and that all locations must be treated equally. Thus, fetching the contents of a memory location always takes the same amount of time regardless of the preceding events. We adopted this principle to prevent the undesirable slow downs we had observed on the Sun 4's with more than 16 active tasks, but this approach does require a very powerful (and therefore expensive) MMU.

As stated before, cache memory adds to the context that must be switched, and is therefore omitted. Another argument against using cache is that it makes performance less predictable. Many real-time applications place as much importance on predictability as on performance. The combined factors of increased context-switch time and lack of predictability excluded most cache designs.

CLOCS is designed to be a complete computer architecture. Simpler computer architectures can handle multiple tasks by combining them into a single program; embedded systems frequently use techniques such as polling to complete several tasks with a single program. However, such systems are expensive to build and maintain, and are often not robust. On the other hand, the CLOCS design includes all of the facilities to run a sophisticated operating system because such capable operating systems are valuable to program development and maintenance. As a result, CLOCS meets the requirement in Fuller's study that a computer must be able to test a new program without endangering any other running programs. CLOCS is an architecture that can support a general purpose operating system.

1.7 The Tradeoff: Throughput vs. Context Switching

CLOCS cannot support faster context switching without compromising elsewhere. For a computer system of the same cost, it is reasonable to expect that CLOCS will have lower throughput performance than an architecture optimized for throughput. The central question is: How big is the degradation? How long must a program run during each activation before the lower throughput of CLOCS overcomes the advantage gained by a faster context switch? If CLOCS has slower throughput but faster context switching than a conventional computer, as long as the tasks run for a long time during each activation, the conventional architecture will activate and run the application in less time, therefore providing more activations per second. But when the application runs for a sufficiently short time, CLOCS will perform better.

Figure 1.2 shows this tradeoff. When the work done by activation of a process is small, CLOCS takes less total time to activate it and do the work. But as the application run time each activation increases, the total time for CLOCS will increase at a steeper rate because the throughput performance of the conventional design is better. At the point indicated by the dashed line, CLOCS and conventional architectures perform the activation and work of the task in the same amount of time. To the right of this line, the conventional architecture is faster. To the left, CLOCS is superior. The point of this study is to find this crossover point. To calibrate Figure 1.2 with numbers requires estimates of CLOCS throughput and context-switch performance. This study provides those estimates. It may be used to evaluate the best kind of architecture for a given application.

Recent advances in integrated-circuit technology may have improved the performance compromise. VLSI implementations allow for large, fast memories to be closer than ever before to the CPU. Although the trench capacitors used in dynamic memory integrated circuits are not commonly used on logic chips such as microprocessors, the most of the other components are the same for all types of digital integrated circuits. Since the memory and CPU fabrication processes are identical or very similar, we can expect little additional relative change in the speed of CPU logic and memory. As a result, today it is more economical than ever before to build computer systems with low latency, high bandwidth memory systems. Although conventional designs also benefit from such memory systems, this implementation technology makes CLOCS

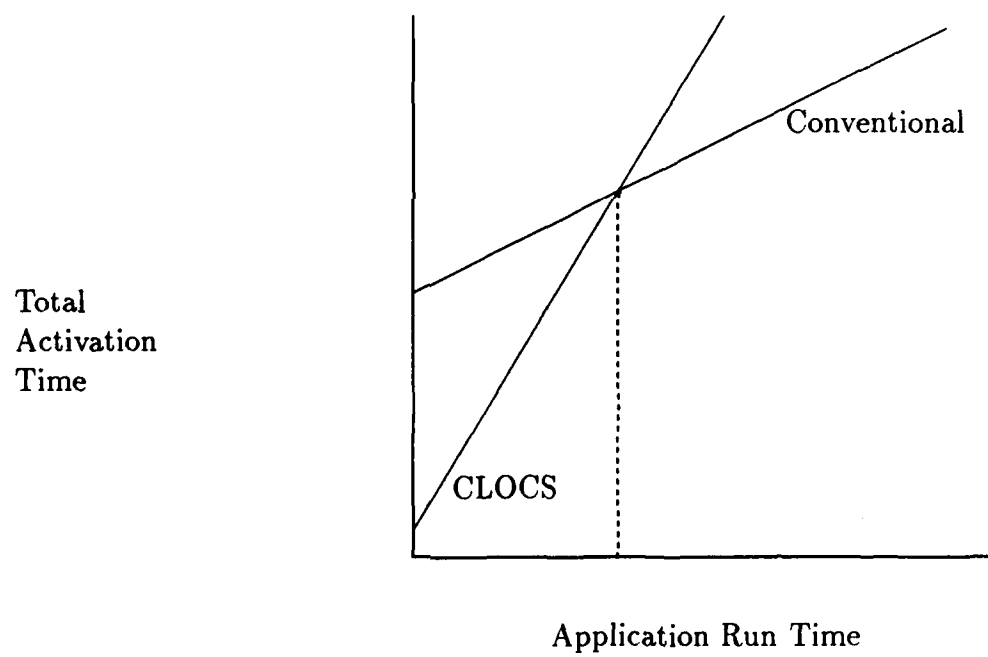


Figure 1.2: Comparison of CLOCS and Conventional Architectures

potentially more competitive. For example, CLOCS with a dual ported, zero wait state memory system could attain one instruction every two memory access times. This compares favorably with the one instruction every 1.2 to 1.4 memory access time of a contemporary Reduced Instruction Set Computer (RISC)[26].

However, finding reasonable tradeoff points confidently requires more than defining an architecture. Estimates of performance must be based on computers that can be designed and built. To get better estimates of performance, the CLOCS architecture has been defined, and an implementation designed and simulated. The implementation is a register transfer model, giving some estimate of the complexity and expense of required components. The execution of programs written in the C language[25] are simulated to determine the number of implementation clock cycles required for the program to run. Performance estimates may then be made by estimating clock rates for a realization of this implementation.

1.8 Summary and Layout of Research

A computer with reduced memory hierarchy will be best for certain applications requiring frequent context switching. Bill Gallmeister and I started this research by reviewing the previous work that is summarized in Chapter 2. Next, we identified the features not directly related to context switching that a computer architecture should include. Those features are described in Chapter 3. Then we designed the CLOCS architecture, and Chapter 4 contains details on that architecture. During the design, we found that a complicated MMU was needed, and it is described in Chapter 5. Also in that chapter is a description of alternate MMU designs that we considered but rejected. Once the architecture was designed, I performed the quantitative analyses of the architecture that are reported in Chapter 6. To obtain better estimates of real-system performance, I designed the implementation described in Chapter 7. This chapter also contains discussion of implementation designs that I discarded. After the chosen implementation design was completed, I simulated benchmark programs on it as reported in Chapter 8. My observations, conclusions and ideas for future work are in Chapter 9.

Chapter II

Previous Work

Computer architects have been designing microprocessors and other general purpose central processing units (CPU) for many years. This chapter summarizes the previous work most closely related to the development of the CLOCS architecture. The RISC approach to computer architecture design has heavily influenced the design of CLOCS. Early RISC work includes the design of the 801 by IBM and the RISC by the University of California at Berkeley. These early designs specified the principles of RISC, but did not develop into commercial products. The RISC principles affected the design of later architectures, many of which became commercial products. This chapter will also examine the Mips Company R2000, the SPARC from Sun Microsystems, and the CRISP from AT&T. Some very recent commercial products are also summarized.

CLOCS also has some similarities to older architectures. Particularly notable are the Atlas from Manchester University for its virtual memory and the Texas Instruments 9900 for its memory-to-memory architecture.

For each of these architectures I will address major contributions and identify problems with context switching or general purpose operating system support.

2.1 IBM 801

Radin reported on the IBM 801[34], a research machine designed and prototyped at the Thomas J. Watson Research Center in the late 1970's. This machine was based on principles that have become important if not essential to the definition of RISC. The IBM 801 was the first RISC and defined the class. The objectives of the design were the following:

- Every effort was made to move hardware functions to software.
- Each feature was evaluated for frequency of use versus cost.

- If possible, actions were moved from run time to compile time.
- The instruction set was chosen to supported the needs of compiler writers.

The 801 had 32 general purpose registers. This computer used a fixed 32-bit instruction format with room for two operand register number and one target register number. All numeric operations used this instruction format, so data had to be loaded from memory before each operation, and the result had to be stored after calculations. This approach to operands was frequently used by later RISC designs and has come to be called a *load/store* architecture. The 801's optimizing compiler used the then newly developed register-coloring[6, 5] algorithm to assign values (expressions or variables) to registers and took advantage of this relatively large number of registers.

The 801 was highly dependent on separate instruction and data caches for high performance. It used a

“store-in-cache” strategy (instead of “storing through” ...)

and the cache was architecturally visible to allow software to manage the cache as much as possible. (The 801 also had some of the data bus control exposed architecturally, but that is not really germane to this research.)

The 801 was designed for rapid interrupt handling. The designers observed that suitability for real time applications depends both on cycle time and on interrupt handling time. Radin's paper noted that IBM did not have any multiuser data and this architecture did not provide extensive support for multiuser operating systems.

The 801 was clearly designed for high throughput. It was not clear that a multiuser operating system would run well on an 801. In any event, high context-switch rates would have been very inefficiently processed because saving all 32 registers would be time consuming. Also the time required to flush the write back cache, would significantly delay context switch because all changed values would have to have been written to main memory.

2.2 Berkeley RISC

Patterson, Sequin, and Katevenis designed the Berkeley RISC[32, 33, 23] with the major objective that an entire microprocessor could be implemented on one chip. This was accomplished by keeping the instruction set and arithmetic logic unit (ALU) as

simple as possible. Like the 801, this was also a *load/store* architecture. To meet a design goal of fast context switching, the Berkeley RISC used registers files. The large number of registers in the CPU might have been divided between processes, and this allowed state for several processes to be in the CPU at one time. The register file may have contained data for up to eight processes, but that was too small a number for a general purpose operating system like UNIX.

Perhaps of greater importance to the designers was the paradigm of the register files used as register windows for subroutine calls. During a subroutine call, the registers visible to programs were shifted in the register file. This allowed some registers to be used for parameter passing, and others to be automatically available for local use. No register savings was required during subroutine calls, so inter-subroutine register allocation is not necessary. Register windows simplified the register allocation task of the compiler by automating it in hardware and requiring the operating system to handle the case that the hardware has used all available register sets.

Using the register files for both context switches and procedure calls would more rapidly exhaust the register file, so the significant contribution of the register files in the Berkeley RISC was the concept of register windows. If the registers were used for register windows, a larger number registers must be saved and reloaded on a context switch. Using register files to support different contexts reduced the average context-switch time, but it did not reduce the maximum switch time because the context for the process that needs to be serviced next (and rapidly) may not be one of the ones in the CPU.

In summary, the large number of registers provided register windows that simplified register allocation during subroutine calls, resulting in better throughput. but increased context-switch time.

2.3 AT&T CRISP

The CRISP machine[10] is an implementation of the C Machine [11] architecture. The architecture is designed to optimize execution of programs produced by the C compiler. The architecture is mostly memory-to-memory, but it does have an accumulator and a stack.

The implementation introduces much additional state into the CPU. For example, there is a stack cache of 32 words. This cache greatly speeds throughput, but can be a major liability during context switching. During context switching, the context of the

process in the stack cache must be saved. Additional performance will be lost as the new process loads values into the stack cache. The implementation also has about 10 pipeline stages. This large number of stages greatly complicates interrupt handling and increases external interrupt latency, thus delaying context switch. Furthermore this long pipeline challenges the designers when solving data dependency problems (trying to fetch a value before it is computed), but they found a solution.

This architecture was not optimized for low context-switch time. The necessity to save the stack cache and the long pipeline make saving state almost as cumbersome as for the Berkeley RISC. This architecture does demonstrate that memory-to-memory architectures support language processors very well.

2.4 MIPS Company R2000

The MIPS company designed and markets a RISC microprocessor called the R2000[22] (and its successors the R3000 and R6000) based on the Stanford MIPS[18, 19, 7] design. This CPU has only one set of 32 registers. Like the 801 and the Berkeley RISC, this is a *load/store* architecture. The context-switch time required to save and restore these registers is increased by the presence of an architecturally visible (and thus mandatory) cache memory, which is a crucial component of the machine's high performance. As the active processes overflow the cache, poor memory system performance will result. The MMU also has provision for a hardware process identifier, but only six bits are allocated to this field, limiting the number of active processes to 64.

This processor is used in the DECStation 3100. I have used it for comparison with a similar CLOCS implementation in Chapter 8.

2.5 Sun Microsystems SPARC

SPARC[15] (for Scalable Process ARChitecture) is a RISC with many similarities to the Berkeley RISC. The machine has much CPU state: the first announced model had 192 registers. Like most of the other RISC designs, it is a *load/store* architecture. Following the Berkeley RISC, the processor has a large number of overlapping register files to optimize procedure calls. The number of files is realization-dependent, and the first version has seven files. These files may be used only as register windows for subroutine calls; there is no ability to split the register file between different pro-

cesses. This large amount of state seriously slows context switching. SPARC is used in the Sun 4 computer, and that computer uses a MMU that has a limited number (16) of user contexts of addressing. If a processes that does not have it addressing context in the MMU is activated, context-switch time will be almost three times longer to complete adjustments to the MMU. Under normal circumstance when just few processes (five is the highest we have observed under normal conditions) are active, this method is very efficient. The large number of registers to be saved/restored and the MMU design of the SPARC/Sun 4 result in context-switch performance only slightly better (two times) than a VAX 780, while its throughput performance is at least seven times better than the VAX 780. This makes context switch marginally acceptable until more that 16 processes are active, then the system performance is greatly degraded as was discussed in Chapter 1. With a sufficiently high number of active processes (about 64) almost all of the CPU time is consumed by context switching.

2.6 Other New Architectures

Other companies have recently released RISC architectures. These architectures do not include any features not discussed above.

The Hewlett-Packard Precision architecture seems to be roughly equivalent to the MIPS R2000, with added support for specific data types (such as decimal for COBOL support). I have no information on how its cache works. In their product information they mention that context switching is adversely affected by the number of registers to be saved (32, as with the R2000). This architecture appears to have similar throughput strengths and context-switch weaknesses as the R2000.

The Motorola 88000 is another new architecture. It also has 32 registers. The on-chip floating-point support uses separate registers that also add to processor state. This microcomputer must use custom designed cache memory chips to form separate Instruction and Data caches. Fast context-switch was not a priority in this design, and, based on the design of the caches, the context-switch performance of this machine is likely to be poor.

The AMD 29000 is a RISC architecture with performance similar to that of the architectures cited above. It has a large register file, with 192 registers that must be saved for a context switch. The saving is speeded by an instruction that saves multiple registers, but still requires a significant time to store all of the registers. This

appears to be the only commercial RISC that retains the Berkeley RISC capability to share the register file among several contexts. This feature is useful in the medium-size controller market that AMD is supporting with this chip, but, like the Berkeley RISC, supports too few contexts for a general purpose operating system.

2.7 Texas Instruments 9900

The Texas Instruments (TI) 9900[21] is a microprocessor designed in the mid 1970's based on the architecture of the TI 900 series minicomputers. This architecture is important to CLOCS because it was a memory-to-memory architecture. The TI 9900 uses the concept of registers as an address abbreviation method, with a pointer in the CPU that located the base of this address space. Programs may address 16 registers (that are actually located in main memory) using this method. CPU state consists of only three words: a program counter, a workspace pointer, and processor status. This machine is very strong on task switching. For example, the three words of CPU state are automatically stored when servicing an interrupt.

This machine suffers from the technology available at the time it was designed. The word size is 16 bits, and only 16 bits are available to form a memory address, limiting total memory size to 64Kbytes. The small word and instruction size made the registers-in-memory paradigm mandatory, but 16 registers are probably not enough for good compiler utilization, particularly since several are used by the CPU hardware. (For example, on interrupt, the three words of CPU state are stored in register 13, 14, and 15. Call and return instructions also use two of these registers.)

Also, there is no support for virtual memory or protection. Although this architecture has great task-switching performance, it cannot support modern virtual memory operating systems or software.

2.8 Atlas

The Atlas[13, 36] (designed at Manchester University in the late 1950s) could perform about 500,000 instructions per second. The memory accesses were for 48-bit words or 24-bit half words and some operations dealt with 6-bit characters. The memory system included 16,000 words of core memory, 96,000 words of drum memory and revolutionary virtual memory hardware and software. The Atlas computer divided the core memory into 32 pages of 512 words each. A page address register was as-

sociated with each page of core memory. During memory operations, the contents of each of these 32 equivalence registers was compared with the high order address bits to determine *equivalence*. *Equivalence* meant that the desired page was in core memory and could be directly accessed using a page number provided by the equivalence registers. These page address registers also maintained a use bit, and every 1024 instructions, special hardware used that bit to calculate the time since loading and time since last access for each page of core memory. A lock bit for each page indicated the page reference was invalid. This was used to prevent access to a page while data was being rolled in from the drum or during other Input/Output. Some consideration for an operating system was included, but this was not really designed to be a multiuser machine. The address space had to be shared between all programs and subroutines and the only provision for protection was the lock bit.

This virtual memory system meets the requirements set forth for the CLOCS MMU. The CLOCS MMU must contain registers similar to the *equivalence* registers.

2.9 Conclusions

Previous work in microprocessor design has advanced the art of RISC to produce high throughput. Studies have shown that increasing state will improve throughput but reduce context-switch performance[12]. Although designs such as the TI 9900 have shown that memory-to-memory architectures are possible, these architectures have not been investigated recently. Thus, CLOCS explores a potentially fruitful area.

Chapter III

Design Considerations Other Than Context Switching

The principal design objective for CLOCS was fast context-switch time. The important tradeoff was execution speed for context-switch time. Chapter 1 sets forth the strategy, principally the flattening of the memory hierarchy, by which the objective is pursued.

Fast context switching is not enough. For example, the Texas Instruments 9900 is a machine that switches context rapidly, but the architecture has a limited address space (by today's standards) and does not support sophisticated operating systems. This architecture does not support with hardware the virtual memory and protection features that powerful operating systems require. What one needs today is a computer that can support the fast context-switching requirements of real-time applications such as process-control and at the same time serve as a base for development of those applications.

Combining a general purpose operating system with real-time, process-control applications has normally been done with two different architectures: a controller and a host system. The objective of this research was to determine if a single architecture, optimized for context switching, would provide an integrated platform to make development easier and supply adequate throughput price/performance. Simplifying the architecture to apply to only part of this environment (for example, process-control) would merely repeat previous research. The purpose of CLOCS research was to investigate a computer architecture to support new applications.

Thus, the applications for CLOCS added some requirements to the architecture. CLOCS must support challenging real-time applications that have frequent context switching among large numbers (hundreds or thousands) of processes. The CLOCS system must provide a development environment with a sophisticated general purpose operating system, compilers, editors, and debuggers. Even when these requirements were recognized, many design choices remained. Whenever possible, we opted for

the simplest hardware-software system that would allow the central thesis to be tested. We considered this approach to computer design to be in keeping with the RISC design philosophy even though the resulting architecture does not resemble other current RISC architectures. Hence machine properties that would substantially reduce the software requirements without compromising the principle that is under investigation were chosen, whether or not they might have been best in some absolute sense. Additionally, we assumed many architectural requirements or support software capabilities to aid in decision making concerning the design. These assumptions fell into four categories:

- Features needed to support general purpose computing
- Operating system support features
- Compiler support features
- Compiler properties to support the architecture

The remainder of this chapter describes these four categories.

3.1 Features Needed to Support General Purpose Computing

To properly execute applications and systems programs, CLOCS must provide an adequate address space, sufficient numeric precision, and an adequate variety of data types. CLOCS must provide a regular (symmetric and orthogonal) instruction set, but each implementation may require the emulation of some instructions.

3.1.1 Adequate Address Space

The amount of memory that computer programs require increases each year [1, 20]. We decided that to anticipate safely memory requirements of the next decade, significantly more than one gigaword (one billion 64-bit words) would be required. We selected that size as a starting point because the address space of the VAX series of computers (one gigabyte) was adequate at the time we did the design, and enlarging the memory by a factor greater than eight seemed adequately conservative. CLOCS provides enough address space for each process to access one terabyte or 128 gigawords of main storage.

In order to provide this large address space without registers, CLOCS uses a form of address abbreviation. Since the largest common program that either designer was familiar with was less than two megabytes, limiting direct addressing modes to 16 megabytes was an acceptable compromise. To abbreviate addresses, CLOCS uses a 24-bit offset in the instruction and a 16-bit segment identifier. Default instruction and data segment identifiers are provided for each process. For indirect addressing, the abbreviation is not required, but the default segment may be used if desired. As a consequence, a CLOCS program may address 16 megabytes directly or one terabyte using indirect addressing. This address abbreviation and memory segmentation does not affect the efficiency of C language programs because the compiler used indirect addressing frequently. Because variables are dynamically allocated, their addresses must be calculated during program execution. As a result, C language programs use indirect addressing for data accesses, and the segmentation scheme does not limit the data space available to programs. Consequently, the C language program only sees the segmentation of the address space as a limit on program size. For example, a program could specify an array of one million by one million characters without causing addressing problems.

3.1.2 Adequate Calculation Precision

For most real-time problems, 64-bit integers and floating-point numbers are sufficiently large. There are no significant architectural reasons to prevent adding decimal or larger floating-point arithmetic, although such features would violate the RISC propriety of the architecture.

3.1.3 Adequate Variety of Data-Object Sizes

During design of the computer architecture, we highly valued simplicity so having only one data-object size (a full-word fixed-point number) was attractive. This restriction permitted a simpler arithmetic logic unit and memory system. However, the benchmark applications to be run on the machine required additional sizes of data objects.

The initial design for the architecture provided only 64-bit data objects, with some small support for 8-bit characters. Analysis of code produced by a prototype compiler revealed that performance on the benchmark programs would be unsatisfactory. In this case, the simplicity of a single data-object size directly affected the

evaluation of the principle under investigation, so the architecture had to be modified. Consequently, we incorporated full, regular support for several sizes of data objects in the design. The most obvious different-sized object was the 8-bit character. The benchmark programs also used 16-bit and 32-bit integers.

Data-object size is indicated in manner orthogonal to operation specification. Since a field is reserved in the instruction for the data-object size, all operations apply to all data-objects sizes. For example, a conditional branch may examine an 8-bit, 16-bit, 32-bit or 64-bit quantity.

3.1.4 Expensive Operations to Be Cheaply Emulated

A regular instruction set was an important design goal. As a result, all the usual arithmetic operations were included in the architecture, even though their implementation would be expensive. Consider the example of divide. The divide operation was included for all sizes of integer data. Implementing this feature in hardware in a microprocessor would be very expensive in area on the chip. However, an implementation may not incorporate the hardware. Instead, executing the divide instruction causes the interrupt for undefined instruction, and the operating system runs a sub-routine to produce the quotient. Because this form of emulation works through context switching on CLOCS, very little overhead is required! Divide is a good example because the divide instruction is infrequently used by most programs[20]. Therefore, most programs will not run much slower, and the silicon area or other resources necessary for divide may be allocated to other purposes. Note, in this case, that the combination of the microprocessor chip and the operating-system emulation routine satisfy the requirements of the architecture.

The design excludes unsigned arithmetic and comparisons. Porting a C compiler would have been much easier with them, but we believed that such an addition would violate the propriety of the design. The unsigned instructions are quite irregular because many of them are the same as signed instructions. Also, we believed that we had already added enough data types and were reluctant to add more.

Floating-point arithmetic is a similar example. Having floating-point hardware might be worthwhile, but may not be feasible in all implementations. However, reserving four operation codes for floating point operations was inexpensive and maintained architectural regularity. Floating-point representation and arithmetic follow the IEEE 754 standard. Since this standard does not conveniently support 8-bit and

16-bit floating-point quantities, floating-point instructions specifying these smaller data-object sizes cause undefined-instruction exceptions.

3.1.5 Predictable Worst Case Performance

An important issue to many real-time applications is the worst-case performance. For these applications, the average time a program takes to run is inconsequential; the maximum time required to run is the only parameter that matters. The system must be designed to get the task completed in a specified time in every instance, or catastrophic failure may result[28]. As a result, architecture or implementation features that improve average performance at the expense of worst-case performance are unacceptable. This consideration eliminated architecturally-visible cache, and influenced the use of pipelining in the implementations.

3.2 Operating System Support Features

Supporting a powerful general purpose operating system is an important goal of the CLOCS architecture. We, the architects, had experience with several operating systems. Our experience ranged from writing UNIX device drivers, to using low-level system calls on MS/DOS systems, to simply using CMS on a VM system. We were prejudiced: we believed that UNIX offered necessary and sufficient features to support a development system, so our definition of a powerful and sophisticated operating systems includes many of the features found in recent versions of UNIX. Of course, current implementations UNIX do not have the required context switch-time performance and most do not support real-time applications at all, but UNIX does supply a list of interesting features. Other facilities for interprocess communication and multiprocessing were also considered to be important operating-system services. To support such an operating system, the architecture had to perform certain memory system and control facilities efficiently.

3.2.1 Virtual-Memory Support

Virtual memory is perhaps the most crucial requirement for running a powerful operating system. A good operating system for a computer system for program development must offer the large address space and the protection that virtual memory affords. Processes must be able to address more data than can be loaded into the

physical memory currently available for that process. Also, the memory space of each process requires protection from the actions of other processes. For virtual memory to run efficiently, the architecture must provide certain services, such as address translation.

3.2.2 Shared-Memory Support

As we examined real-time systems and implementations of real time systems on UNIX computers, we noted that shared memory was considered important or critical to many real-time applications. Since these applications are important to CLOCS, the memory management scheme includes a shared memory facility, so that the same physical memory may be read and modified by different processes.

3.2.3 Classes of Main-Storage Areas

Closely related to virtual-memory support is the support of different classes of main-storage areas. For example, a program deserves protection from itself, so that a program cannot inadvertently modify itself or inappropriate portions of its data space. Also, reserving a part of the address space for communication improves the efficiency of operating system service requests.

3.2.4 Semaphores

The best way for an architecture to support multiprocessor synchronization and efficient interprocess communication is with an atomic operation such as test and set. Although supporting semaphores and other synchronization techniques was not a high priority, it was a factor in some decisions, such as the inclusion of conditional skips.

3.3 Compiler Support Features

In modern computer systems, almost all programs are generated by a compiler. Only a few specialized subroutines, such as character-string move subroutines or operating-system memory-management operations, are coded in assembler or machine language. As a result, the compiler must be able to produce efficient machine language programs from the input programs, written in the C language in this case. As we designed the

architecture and ported two C language compilers to the architecture, we identified several important features, notably multiple data object sizes and addressing mode capabilities.

3.3.1 Data-Object Size

As described above, regular, efficient support for multiple sizes of data objects noticeably affects performance of the benchmark programs. Having this size flexibility is important for the compiler also. The compiler is greatly simplified if it can treat different-sized objects similarly for both operations and addressing.

Consider the example of operating on two 8-bit characters. If the characters must first be extracted from a 64-bit word, complexity will be added:

1. Instructions must be generated to extract the character.
2. Temporary storage must be allocated to hold the characters.
3. The operation must be completed.
4. The result must be stored back into some word.

More serious than the extra work is the problem of addressing. If the characters have a two-part address, the address of the word and the address of the character within the word, the address is a different type from the addresses of a full-word integer. Multiple address types adds great complexity to the compiler.

These problems can be solved. The C language compilers for the Data General MV series and the CRAY 1S are good examples of solutions. However, the increased difficulty in compiler writing makes the two part address approach inferior for CLOCS. Also, this approach imposes a performance degradation which is amplified by the nature of the programs we selected for benchmarks. As a result, we added the design consideration of multiple data-object sizes with consistent addressing based on the size of the smallest object, the 8-bit byte.

3.3.2 Addressing Mode Expectations

Compilers require the ability to modify operand addresses. In the C language, arrays, pointers, subroutine calls and automatic allocation of variables all require the calculation of addresses at program run time. This capability is provided through indirect addressing.

Although both of the compilers used with the CLOCS architecture [24, 37] assume the ability to add small constant offsets to calculated addresses, this useful feature is too expensive of instruction address bits to incorporate into the CLOCS architecture.

3.4 Compiler Properties Required to Support the Architecture

A major part of the CLOCS design was based on the assumption that the C language compiler would generate machine language programs that took advantage of the architecture and were well optimized. That is, the compiler would efficiently use the instructions that CLOCS provides and it would not generate unneeded or inefficient instruction sequences. We assumed that as long as the instruction set was regular and consistent, the compiler would be able to produce near-optimal machine code. Some features of modern compilers, such as a high-quality register allocator, were not required. This architecture does demand some important compiler features, though.

3.4.1 Compiler Optimization

Indirect addressing results in many explicit calculations of addresses, which in turn results in much opportunity for common expression elimination. These common expressions may span several basic blocks. For example, an automatic variable may be fetched several times during a program, and each reference involves calculating the address. The compiler must be intelligent enough to eliminate a significant fraction of the redundant address calculations.

3.4.2 Support Different Data-Object Sizes

The compiler for CLOCS also needs to support 64-bit integers and 64-bit word sizes. The size of other data types must be independent of word size. For example, if the compiler assumes that four characters could be placed in each word when actually eight will fit, significant problems arise. Many of the portable compilers we examined or used had such a limitation. The GNU C language compiler, even though it claims to support only a 32-bit word size, is relatively easy to modify to add the 64-bit word operations. The compiler must understand the alignment restrictions of the architecture, and the GNU C compiler had this feature.

Chapter IV

Description of the Architecture

This chapter presents the architecture of CLOCS. It begins with a survey of the architecture highlights, then covers instruction formats, programming model, data formats, and operations. A section on implementation concludes the chapter.

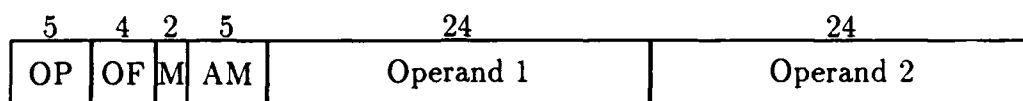
4.1 Highlights

The CLOCS architecture aims to reduce the effort of switching execution from one task to another by removing the highest layers of memory hierarchy. As a result, the CLOCS architecture is a very simple one, with only memory-to-memory instructions. The program context includes a single status register containing a program counter, a process identification number and various flags.

This machine was designed using applicable RISC concepts. For example, all instructions are 64-bits long. Data objects of eight, 16, 32 and 64-bits are supported, but these objects must be strictly aligned (e.g., a 32-bit object must start on a 32-bit boundary), and all instructions start on 64-bit boundaries. These definitions of length and alignment permit the memory system implementation to deal only with 64-bit words. This simplification greatly improves memory system performance[32].

4.1.1 Noteworthy

The machine can switch context in less than the time required for the processor to execute one instruction, since only the time for one store to memory and one fetch is required for the switch. The result is much higher performance in programming environments with frequent context switching; the requirements of many applications for very fast context switching are satisfied.



OP Operation Code

OF Operation Flags for skip and branch

M Mode (size of operands)

AM Address Mode for Operand 1 and Operand 2

Figure 4.1: CLOCS Instruction Format

4.1.2 Peculiarities

The only state inside the central processing unit (CPU) is a program status word. All operations access data in main memory. One consequence of this is that all parameter passing must be through memory.

4.1.3 History

The CLOCS architecture was conceived in May of 1987 by Mark Davis as an idea of how he would design a RISC computer. The architecture was designed by Mark C. Davis and Bill O. Gallmeister in the fall of 1987. In this dissertation, “we” refers to Bill and me.

The architecture was revised in December of 1988 and February 1989 to improve performance on common integer-benchmark programs. The changes did not affect context switching, but did add complexity to the architecture.

In August 1989, I made some additions to the architecture to ease implementation and simulation. These extra operations are discussed in the Chapter 7 on implementation.

4.2 Instruction Formats

There is one instruction format; it is always one word long (Figure 4.1). There are two operand fields. Most commonly, data addressed by Operand 1 and data addressed by Operand 2 are combined using the specified operation, and the result is stored in the location specified in Operand 2.

4.2.1 Address Specification

The 24-bit operand in the instruction may be used as immediate data or may be combined with a 16-bit segment identifier (*SID*) to form a 40-bit virtual address. Each process has a primary instruction *SID* (*OSID*) and a primary operand *SID* (*OSID*) assigned. Indirect addressing is also supported. During indirect addressing, the default *OSID* or *ISID* may be overridden to access any byte in the terabyte (2^{40}) address space.

4.2.2 Variations of Field Use

Although all instructions have the same format, the meaning of some of the fields are different for some classes of instructions.

Operation Flags

The four bits in the Operation Flags (OF) field have the following meanings:

LT The result of the operation is less than zero.

GT The result of the operation is greater than zero.

EQ The result of the operation is equal to zero.

NO The result of the operation did *not* result in an arithmetic exception.

For the *Branch* and *Trap* operations, the first three flags are used to determine whether the trap will occur by evaluation of Operand 2. Since Operand 2 is only fetched for the comparison, no arithmetic exception is possible. For all other operations, the flags are used with the result of the operation (that will be stored in Operand 2) to do a conditional skip of the next instruction.

Operation Modes (Object Size)

The Operation Mode field (M) specifies the size of the operand objects. Sizes of eight, 16, 32, and 64-bits may be specified. In general, this field specifies the size of both Operand 1 and Operand 2. However, for branches, the size of Operand 1 (the destination of the branch) is always 64, so the Operation Mode applies only to Operand 2. To simplify compiler construction, the distance of shifts is always specified by a 64-bit number. As a result, for shifts, the size specified by the OM only applies to the size of Operand 2.

Fetch and Store of Operand 2

The execution of instructions does not always result in the fetching of Operand 2 nor in the storing of Operand 2. Some control instructions (*Branch*, *Trap*, and *Load Operand Segment*) do not change the contents of Operand 2. When those same instructions are unconditional, Operand 2 is not referenced, so need not be fetched.

4.2.3 Spaces and Addressing

The only numbered address space is main memory. The CPU deals with virtual addresses, and these are converted to physical addresses by the Memory Management Unit (MMU). A virtual address has 40 bits, yielding an address space of one terabyte. There are 30 bits in a physical address, for a space of one gigabyte.

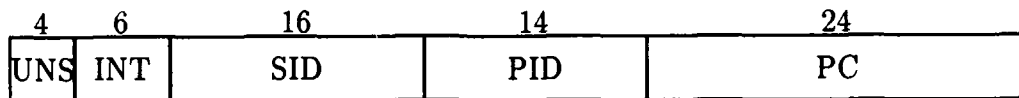
4.3 Programming Model

The programming model is pretty simple. All operations are memory-to-memory. There is no working storage (registers). Everything is kept in main memory. The memory name-space is a linear sequence of 2^{40} 8-bit bytes, and it contains special locations for items such as interrupt vectors and MMU control registers. Even the program counter (Status Word) may be accessed using a memory address. Peripheral devices are also memory-mapped, and so it is important that the memory system properly handles these special addresses. The CPU intercepts reference to the program counter, and the MMU handles references to its own registers. It is the responsibility of the operating system to make sure that other memory-mapped addresses are handled properly by the MMU.

4.3.1 Control Storage

A Status Word, MMU register, and other Input/Output registers constitute the control storage for CLOCS. The Status Word, which is the only task-dependent data kept in the CPU, is a single 64-bit word and is shown in Fig 4.2. It contains a 24-bit Program Counter, a 14-bit Process Identifier (*PID*) used to identify ownership of system resources, interrupt mask flags, and reserved bits. The interrupt mask flag bits have the following meanings if set:

1. Do not interrupt for arithmetic exceptions.



UNS Unused, Reserved Bits

INT Interrupt Mask Flags

SID Segment Identifier

PID Process Identifier

PC Program Counter

Figure 4.2: CLOCS Status Word Format

2. Do not allow group Zero Interrupts.
3. Do not allow group One Interrupts.
4. Do not allow group Two Interrupts.
5. Do not allow group Three Interrupts.
6. Reserved

The MMU contains an implementation-dependent number of one-word registers. These registers provide sufficient information to transform virtual addresses to physical addresses. These registers and the operation of the MMU is discussed in more detail in Chapter 5.

CLOCS reduces the variety of required instructions by mapping all state information of the machine into the memory space of the processor. Thus, the State Word may be found at location FFFF-FFFFFF (This is segment FFFF, address FFFFFFFF in hexadecimal). Other memory locations reserved for special functions include:

- The (131,071) MMU registers begin at FFFF-F00000.
- Location FFFF-FEFFF8 contains the number of MMU registers installed on this CPU.
- Input/output devices are mapped into the 7680 memory locations from FFFF-FF0000 to FFFF-FFEFF8.

- The 511 trap and interrupt vectors, likewise, can be found in this segment, at addresses FFFF-FFF000 to FFFF-FFFFF0. New status word to be loaded on interrupt start at FFFF-FFF000, and are in four groups of 64 each. The status words for traps follow them.

These special locations are virtual addresses. The CPU is responsible for intercepting references to the status word, providing data for read operations, and preventing write operations if the current *PID* is not zero. The MMU is responsible for handling references to the MMU registers. The other locations are translated to physical memory addresses by the MMU, so the operating system must assure that the correct physical locations are specified, and the page swapping routines do not try to *swap out a memory device*. By allowing virtual input/output device addresses and interrupt vectors, CLOCS supports simulating virtual machines.

4.3.2 Address Calculation and Addressing Modes

Because the CLOCS design does not include registers for use with memory addressing, the architecture includes more unconventional methods to access memory. Below are descriptions of the formation of virtual addresses, the transformation from virtual to physical addresses, and the addressing modes available to programs.

Virtual-Address Formation

A virtual address is 40 bits long, and it may be formed in two ways. First, it may be fetched from the low-order 40 bits of a memory word by indirect addressing. More usually, a 16-bit segment identifier (*SID*) and a 24-bit offset (Operand Offset) are combined to form the address. Each process has a default segment assigned for instructions and another for data. The MMU stores these segment identifiers and uses the process identifier to find the correct segment identifier. The 24-bit offsets appear in the instructions or may be fetched from main memory by indirect addressing.

Physical-Address Formation

The MMU can calculate a physical address in one of two ways. In the first case, the CPU provides a process identification number and a 24-bit offset. The MMU associatively looks up the physical page corresponding to the default segment for the given process and the 12 high order bits of the offset. In the second case, the CPU

provides the entire 40-bit address. Then MMU the associatively looks up the physical page corresponding to the 28 high-order bits (16 *SID* + 12 from offset). After the physical page has been identified, the MMU verifies that the requested operation (read or write) is authorized for this process. If it is, the 30-bit physical address is formed from the 18-bit physical page address and the 12 low-order bits from the virtual address.

Addressing Mode Overview

Seven addressing modes apply to Operand 1, and four of these also apply to Operand 2. All modes used for Operand 2 have a high-order bit of zero, so only the two low-order bits appear in the instruction. Listed below are the addressing modes, with the abbreviations used by the CLOCS assembler following in quotation marks, applicability to operands, and a description. In these descriptions, “+” means “concatenate the two values,” “FETCH” means “obtain the contents of main memory at the specified address,” and “:=” indicates the assignment of the value for later use by the CPU.

Direct “Opnd”

Applicable to **Operand1** and **Operand2**.

Operand := FETCH (*OSID* + Opnd)

OSID, the operand *SID*, is concatenated to the high-order end of Opnd to provide a full 40-bit virtual operand address from which the operand is fetched. This is the *direct addressing mode* used by CLOCS to obtain an operand.

Indirect “@Opnd”

Applicable to **Operand1** and **Operand2**.

Operand := FETCH (*OSID* + FETCH (*OSID* + Opnd))

OSID is concatenated to Opnd to form a virtual address. From this address a 24-bit offset is fetched. This offset is concatenated with *OSID* to form the virtual operand address. This is the *indirect-addressing mode* used by CLOCS to obtain an operand.

Zero Page “%Opnd”

Applicable to **Operand1** and **Operand2**.

Operand := FETCH (0 Segment + Opnd)

The operand is concatenated to a *SID* of zero to arrive at the virtual operand address. This provides rapid zero-page addressing, but otherwise is identical to direct addressing. This is the *zero page-addressing mode* used by CLOCS to obtain an operand.

Absolute Indirect “%@Opnd”

Applicable to **Operand1** and **Operand2**.

Operand := FETCH (FETCH (OSID + Opnd))

OSID is concatenated to *Opnd* to form a virtual address; from this address a word containing a 40-bit address is fetched to form a virtual address into any page. This is indirect addressing from the default page into any page. This is the *absolute indirect-addressing mode* used by CLOCS to obtain an operand.

Zero Page Absolute Indirect “%@@Opnd”

Applicable to **Operand1 Only**.

Operand := FETCH (FETCH (0 Segment + Opnd))

Opnd is concatenated with the zero-page *SID* to form a virtual address; from this address in the zero page a word containing a 40-bit address is fetched. This virtual address is used to fetch data in any page. This is indirect addressing FROM the zero page, INTO any page. This is the *zero-page absolute-indirect addressing mode* used by CLOCS to obtain an operand.

Zero Page Indirect “@%Opnd”

Applicable to **Operand1 Only**.

Operand := FETCH (OSID + FETCH (0 Segment + Opnd))

Opnd is concatenated with the zero page *SID* to form an virtual address. From that address, a 24-bit offset is fetched. This offset is concatenated with the *OSID* to form the virtual operand address. This is indirect addressing from the zero page into the default page. We do not see a great need for this instruction, however we put it in for symmetry. It may be of some future use for providing operating system services. This is the *zero-page indirect-addressing mode* used by CLOCS to obtain an operand.

Immediate "<Opnd"

Applicable to **Operand1 Only**.

Operand := Opnd

Opnd is a 24-bit immediate operand. For all operations except floating point arithmetic, the 24 bits represent a two's complement integer. For floating point, the immediate operand may only be zero; all other values are reserved. This is the *immediate addressing mode* used by CLOCS to obtain an operand.

4.3.3 Addressing Mode Summary

Table 4.1 summarizes the types of memory access that occur for each addressing mode. All memory transactions are for 64-bit words. Therefore, the data object size of the instruction is not pertinent during a fetch. The indirect-addressing modes use the low-order 24 or 40 bits (as appropriate) of the 64-bit word retrieved by the first fetch. This is true no matter what the instruction operand size.

The first memory fetch during any indirect-addressing mode always uses the default Operand Segment Identifier (*OSID*). This is necessary so that programs can easily store and move branch addresses; the MMU will not permit writing to the memory space addressed by the default Instruction Segment Identifier (*ISID*). Also, the immediate-address mode does not apply to the instruction space, and a branch with an immediate Operand 1 causes an exception.

4.4 Data Formats

4.4.1 Fixed Point

Integer data are represented as eight, 16, 32, and 64-bit two's- complement numbers.

4.4.2 Floating Point

Floating-point numbers are represented as 32 and 64-bit numbers, conforming to IEEE 754 formats.

Addressing Mode	Type of Memory Access	
	First Fetch	Second Fetch
Direct	Relative	
Indirect	Relative	Relative
Zero Page	Zero	
Absolute Indirect	Relative	Absolute
Zero Page Absolute Indirect	Zero	Absolute
Zero Page Indirect	Zero	Relative
Immediate		

Relative means use the default SID.

Absolute means use the SID provided (0 or from indirect address).

Zero means use zero for the SID.

A blank column means memory access is not required.

Table 4.1: Summary of Addressing Modes

4.4.3 Character

An 8-bit integer may represent a character. There are no other architectural limitations on the character set used.

4.5 Operations

The CLOCS architecture has 20 operations. There are one movement, five fixed arithmetic, four floating-point arithmetic, three logical, four shifts, one sequencing and two supervisory. For ease of compiler writing and simulation, some additional pseudo instructions were invented (such as conversion from fixed- to floating-point formats), but these are not considered part of the architecture. The compiler produced the instructions as if they were supported by the architecture and the implementation-level simulator simulated their execution. Two alternate, more realistic approaches were for the compiler to generate subroutine calls instead of these instructions or for the operating system to emulate the instructions. I did not use either of those approaches because they were more time consuming to implement and would not have improved the accuracy of simulation timings.

4.5.1 Decision

CLOCS has no specific decision operations. Instead, a conditional branch is provided, and all arithmetic and boolean logical operations incorporate conditional skip. The behavior of this sequencing will be discussed with each category of operation.

4.5.2 Data Operations

Data operations are partitioned into fixed- and floating-point arithmetic, boolean logic operations, and shifts.

Arithmetic Operations

CLOCS supports 8, 16, 32, and 64-bit two's complement fixed-point arithmetic. Operations require operands of the same length. For operations such as multiply, with larger results than sources, the high-order bits are lost. Similarly, the fractional part of a divide result is lost. Indication of multiply overflow is available to the programmer in the form of a conditional skip. The program may use the remainder instruction to detect and manipulate fractional divide results.

Operation codes have been set aside in CLOCS for floating-point arithmetic. Early implementations of CLOCS would not include floating-point hardware: floating-point instructions would cause *unknown operation* faults, and the operating system would then perform the floating-point operations. Later implementations could then add floating-point hardware, and unchanged programs would automatically take advantage of the improved performance.

As discussed above, the Flags field of the instruction governs a conditional skip. For both fixed-point and floating-point arithmetic, the following instruction is skipped if one or more of four conditions is flagged in the instruction and that condition is true. These conditions are:

LT The result of the operation is less than zero.

GT The result of the operation is greater than zero.

EQ The result of the operation is equal to zero.

NO The result of the operation did NOT overflow, did NOT underflow, or NOT a divide-by-zero, as appropriate for the operation.

Boolean-Logic Operations

CLOCS provides *And*, *Or*, and *Xor* logical operations.

Skips for boolean-logical operations occur for three possible conditions:

LT The result of the operation is less than zero.

GT The result of the operation is greater than zero.

EQ The result of the operation is equal to zero.

Note that Operand 2 is evaluated as a two's-complement number for the conditional skip; however, the sign and equivalence to zero of 4-byte and 8-byte floating-point numbers are also interpreted correctly.

Shifts

CLOCS provides *Shift Left*, *Shift Right*, *Shift Right Arithmetic* (extends two's-complement sign), and *Rotate Left*. The number of bits to be shifted (from zero to the size of Operand 2) is specified in Operand 1. The following instruction is skipped using the same conditions as the logical operations above.

4.5.3 Sequencing

The sequence of instructions is controlled by the branch instruction, supervisor calls, and interrupts.

Branches

The branch instruction is conditional, based on the contents of the second operand (evaluated as a fixed-point number):

LT Operand 2 is less than zero.

GT Operand 2 is greater than zero.

EQ Operand 2 is equal to zero.

Interrupt and Supervisor Call

As described in the section above on Control Storage, CLOCS has a large number of interrupt vectors. On a Supervisor Call (*Trap*) or an interrupt, the old status

word is saved and the new status word for that supervisor call or interrupt is loaded. Interrupts are grouped into four maskable levels. Each interrupt status word should mask its group of interrupts long enough to move the saved status word out of the way. Then it may re-enable that group of interrupts. In this way, if the same interrupt occurs before processing of the first interrupt is complete, the first interrupt will not be lost.

The *Trap* instruction is the form of supervisor call on CLOCS. Operand 1 specifies the number of the trap which is then used as an index into the trap vector beginning at FFFF-FFFF800. The corresponding trap status word is loaded into the CPU and the old Status word is saved.

The trap instruction execution is conditional. If one of the flags in the instruction is set then Operand 2 is fetched and examined. The *Trap* instruction uses the same conditions as the branch instruction; if the corresponding condition is true, then the *Trap* occurs and the CPU stores the Status Word and loads the new Status Word specified by the *Trap*. Otherwise, the following instruction is executed.

4.5.4 Supervisory

Two supervisory instructions are provided. The *Trap* instruction conditionally causes the execution of a supervisor call at a trap vector location as described above. This qualifies as a supervisory instruction because the status word is directly loaded from the trap vector, allowing the machine to change to operating system process-identification number. Condition flags for this instruction are the same as for the branch.

The *Load Operand Segment* instruction allows a program to use a different default data segment. If any of the flags are set, then Operand 2 is fetched and examined. If a flagged condition is satisfied, the next instruction is skipped. The *Load Operand Segment* instruction is always executed. If the identified segment is not available to the process, the CPU will cause a fault.

Although not specifically allocated as a supervisory instruction, moving data to certain addresses from FFFF-FF0000 to FFFF-FFFFFF causes changes to the computer system. For example, writing to FFFF-FFFFFF changes the Status Word. In the case of the Status Word, the CPU enforces writing only by process number zero. For other memory location, the operating system must use the MMU registers to prevent unauthorized modifications.

4.5.5 Input and Output

Input and output devices are memory-mapped, so no special operations are provided to manage them. The memory-mapping is in the memory-address range FFFF-FF0000 to FFFF-FFEFFF. A special set of addresses is provided to standardize virtual memory and cache algorithms, so they will not interfere with proper device operation.

4.6 Implementation Notes

The architecture leaves two major hooks to permit single chip implementations with a reasonable numbers of transistors. First, operations are defined for floating point, but no hardware support is required. Under normal circumstances, floating-point will be emulated by operating-system routines. The second hook concerns the size of the MMU. Although memory addresses have been set aside for a very large number of MMU Registers, a machine could be built with very few registers, perhaps with as few as four. Although economizing on the MMU will save chip area, it will have a major impact on context-switching performance; therefore, we recommend having at least one MMU register for each page of physical memory installed in the machine.

The implementations of CLOCS can be heavily pipelined. A four-stage pipeline with interlocks or a seven-stage pipeline without interlocks seems reasonable. Note that pipelining will increase context-switch latency, which may be significant if a real-time task has to be serviced in less than 20 cycles. It is not clear how one writes a scheduler for such a demanding environment, but the increased latency is a consideration. Also, caching inside the CPU may effectively improve performance. One type of caching is already available with a pipeline design; the result of the previous calculation is available for use by the current instruction. I expect instructions using the result of the previous instruction to be common (compute address followed by fetch data), so this may be important for good performance.

Chapter V

Description of the Memory Management Unit

This chapter begins with a description of the organization of the Memory Management Unit (MMU). The details of the MMU design and descriptions of operations follow. The fourth section shows how the MMU supports common virtual-memory algorithms. The chapter concludes by describing two MMU designs that were discarded.

5.1 Organization of the Memory Management Unit

5.1.1 Requirements

The CLOCS MMU must support virtual memory for large numbers of active processes without seriously affecting context-switch performance. The guiding principle for MMU performance is: "If information for a process is in main memory, it must be accessed with no context-switch penalty." To support general purpose operating systems adequately, the MMU must provide each process a unique address space. Processes that have their own address space are sometimes called *heavyweight processes*[9]. Another design requirement is that the MMU must support *lightweight processes*. This means that the MMU must provide a sharable address space with protection for the space owned by each process. Provision for protected sharing of memory between two processes is also an important requirement for real-time applications.

5.1.2 What the MMU Does

The purpose of the MMU is to support virtual memory for the CLOCS computer system. For each memory system reference, the MMU does the following:

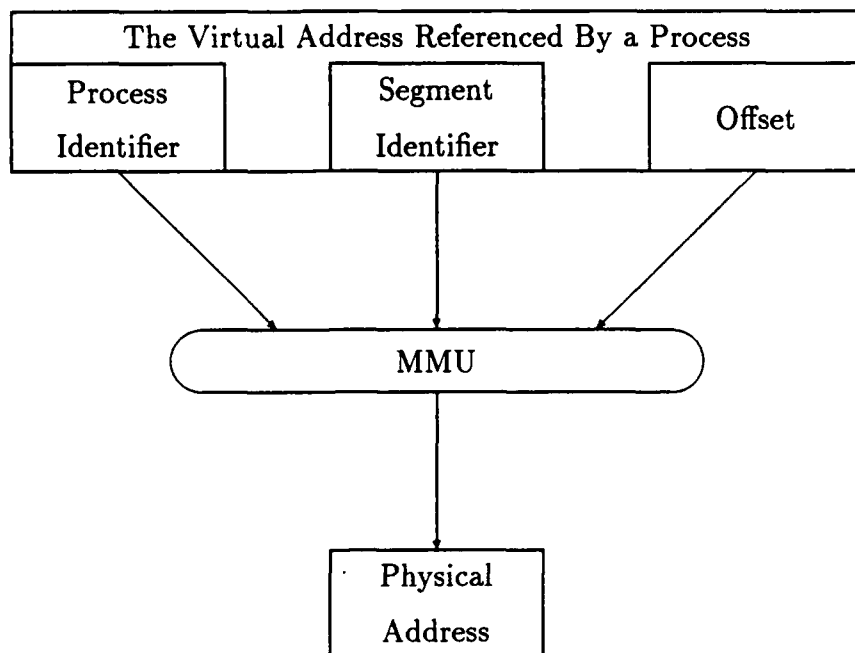


Figure 5.1: What the CLOCS MMU Does

- It accepts an address specification from the CPU.
- It determines the corresponding physical address.
- It verifies permission for the requested memory operation.
- It maintains information of use to the operating system.

To specify an address, the program provides a 24-bit Offset and a 16-bit Segment Identifier (*SID*) which form a 40-bit virtual address. The Offset may come from either one of the operand fields of an instruction or from memory during an indirect address. The *SID* may be a default Segment Identifier for the process, zero, or a *SID* obtained from memory during an indirect address. Because each process must have its own address space available, the process identifier (*PID*) is also part of the address specification. This address translation function of the MMU is shown in Figure 5.1. Although it is very similar to the scheme used in the Atlas computers [36], the Atlas scheme did not have the extensive support of permissions and status that the CLOCS MMU provides.

Not every bit of an address is translated. In CLOCS, a memory page is 4K bytes of either virtual or physical memory. The real work of address translation is done at the page level; the low-order 12 bits of the virtual address are used as the low-order

bits of the physical address. The 28 high-order bits of the virtual address form the virtual page.

The CLOCS MMU supports three types of operations: read, write, and execute. It allows the operation only if the addressed virtual page has the appropriate authorizations for the current process. Authorizations for a virtual page may be one of the following:

- No operations authorized for this page
- Read operation only
- Read or write operations
- Fetch instruction operations only

The MMU also keeps records of access and writing to physical pages. It records if a physical page has been read and written, a state called *USED*, so the operating system can later determine the best page to swap out using a well known algorithm for virtual memory[9]. The MMU also records if a *read or write* type physical page has been changed, a state called *DIRTY*, so the operating system can avoid unnecessary saving of pages to backing store. The encoding and manipulation of this information is discussed later in this chapter.

5.1.3 External Appearance of the MMU

All the information to do the above — determine physical addresses, check permission, and remember physical memory status — is kept in 64-bit registers in the MMU. Each register relates one *PID* and virtual page to one physical page. These MMU registers are memory-mapped, beginning at location FFFF-800000, and are specially protected from user processes. Only the superuser, *PID* = zero, may change them. To support the guiding principle of equal time access for all data in memory, the MMU must contain at least as many registers as the computer system contains physical pages. More registers must be provided, because one extra MMU register is required for each page of memory shared between two processes. In the absence of data for specific applications, we estimate that an additional 10% of MMU registers over the maximum number of expected physical pages is sufficient. CLOCS can address up to 256K physical pages (2^{18}), but since this corresponds to one gigabyte of memory, most machines will have less physical memory and need many fewer MMU registers.

Memory address FFFF-FEFFF8 is reserved for a 64-bit fix-point integer representing the number of MMU registers installed. When this location is read, the MMU responds with the number of MMU registers installed. This same memory location is also used as a command register. Writing to this location causes special MMU operations such as resetting the *USED* flags for all pages, as described later in this chapter.

5.1.4 Physical Page Status

One function of the MMU is to maintain status information on the memory pages, so a part of each MMU register contains bits indicating the status of the corresponding virtual and physical page. The statuses are not orthogonal, so an encoding scheme reduces the number of bits needed to indicate the status. The *USED* and *DIRTY* statuses are maintained for the physical page. More than one MMU register may refer to a physical page; this allows memory to be shared. In this case, the MMU must provide the correct status for a physical page when any MMU register referring to that physical page is read. For example, suppose MMU register FFFF-F00010 and FFFF-F00090 both point to physical page four. A write is made using the entry at FFFF-F00090. If the register at FFFF-F00010 is subsequently read, its status will indicate that the page is *DIRTY* even though no write was made using that MMU entry.

The implementer may use any means to accomplish this subtle updating of physical-page status, but one solution is suggested. An auxiliary memory with a 2-bit word for each physical page stores the correct status of each physical page. During routine memory operations, the status of a page is updated concurrently parallel with the memory operation. When an MMU register is read, the physical-page address in the MMU register is used to access the auxiliary memory. The *USED* and *DIRTY* bits from the auxiliary memory are used to update the MMU register before it is provided to the CPU. As long as the page status is fetched and the MMU register status updated in the time of a main memory fetch, the MMU organization does not affect performance.

5.2 Contents of the MMU Word

Before we examine the MMU registers, here is a quick review of terms describing CLOCS memory addressing and MMU register fields. Each abbreviation is followed by the number of bits allocated in this system.

VA (40) Virtual address

PA (30) Physical address

PID (14) Process identifier

SID (16) Segment identifier

OSID (16) the primary (or default) *SID* for operands

ISID (16) the primary (or default) *SID* for instructions

VO (24) Virtual Offset (the address contained in instructions)

PC (24) Program Counter, a *VO*

OPND (24) Operand address in an instruction, a *VO*

VP (12) Virtual-page address, the high order bits of the *VO*

PP (18) Physical-page address

PO (12) Physical Offset, the low-order bits of *VO*

FLG (4) Flags, indicating permissions and status of a virtual page

The MMU takes the *VA* (made up of a *SID* and a *VO*) and the *PID* and produces a *PA* that is 30 bits long. The translation process is shown graphically in Figure 5.2.

Each MMU register (or entry) is a 64-bit word. The MMU registers are divided into six fields. The MMU may store the information for each entry in any convenient format, but it must appear as a 64-bit memory word with the format shown in Figure 5.3 when read by the CPU.

5.2.1 Sizing Considerations for Fields

We determined the size of each of the fields in the MMU registers in two iterations through the design process. First, we determined the most desirable size for each

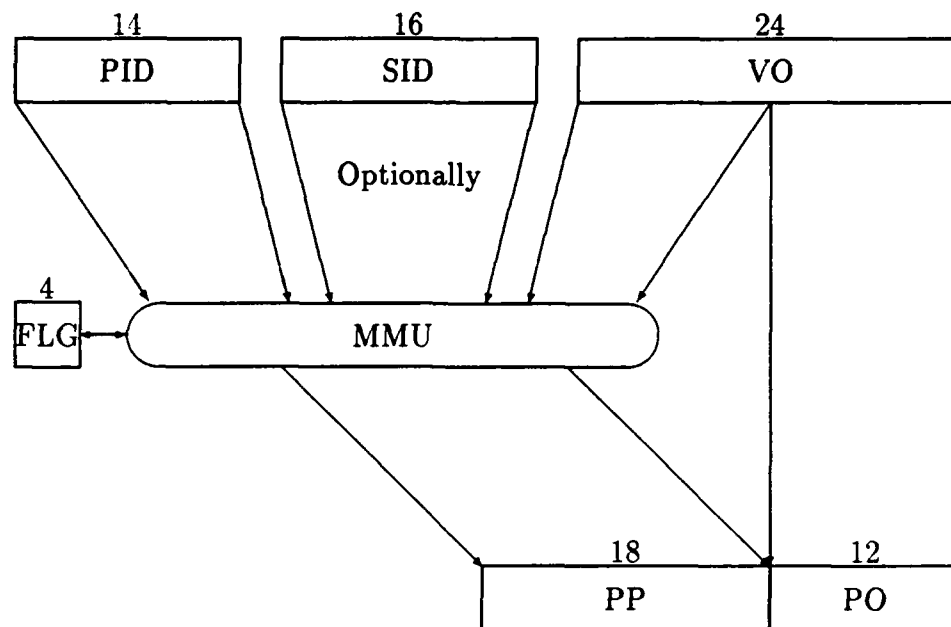


Figure 5.2: CLOCS MMU Address Translation (Detailing of Figure 5.1)



PID 14 bits - Process Identifier

FLG 4 bits - Permissions and Physical Page State

SID 16 bits - Segment Identifier

VP 12 bits - Virtual Page

PP 18 bits - Physical Page

Figure 5.3: CLOCS MMU Word Format

field. As frequently happens, there were not enough bits available to satisfy all us, so we made some compromises and solved some simple inequalities to make the field size assignments. We started from some quantities fixed by previous architecture design decisions, then adjusted the other fields using the considerations mentioned as guidelines in Chapter 3.

As we began the process of setting desirable field sizes, we had one fixed starting point: the size of the virtual address offset (*VO*), which occupied 24 bits in the instruction. In order to make the address space sufficiently large, we set the size of the *SID* to 16 bits. We wanted at least one gigabyte of physical storage, so the physical address required 30 bits. *FLG* required about four bits. We wanted 16 bits for *PID* adequate to support 64K processes, a number that was more than sufficient for the UNIX systems we had studied. We set the physical page size to 1024 bytes based on our opinions concerning data object sizes. This size represents 128 words, a quantity small enough to allow very complete use of available physical memory. Since the *VO* was 24 bits, this physical page size would have required the *VP* to be 14 bits and the *PP* to be 20 bits. The 34 bits for *VP* and *PP* plus the 16 bits for *SID* leaves only 14 bits for *FLG* and *PID*. We estimated that at least four bits would be required for *FLG*, leaving only 10 bits for *PID*.

Now it was time for us to compromise. Since the 64K processes limit available under UNIX was much larger than needed, the *PID* was a good place to start looking for some additional bits. A 10-bit *PID*, allowing only 1024 active processes, was too small to meet the design consideration of handling thousands of processes. Instead we accepted a 14-bit *PID*, allowing 16K processes. This total of 34 bits for the *PID*, *SID*, and *FLG* left 30 bits. Then solving the constraints:

$$VP + PP = 30 \text{ bits}$$

$$PP + PO = 30 \text{ bits}$$

$$VP + PO = 24 \text{ bits}$$

we obtained the final values listed in Figure 5.3. The 12 bits allocated to the physical offset resulted in a physical page size of 4K, which was larger than we desired, but a size commonly used for UNIX virtual memory systems. This compromise increased the importance of maintaining the *FLG* field no larger than four bits, so we minimized the number of bits required, as discussed below.

			D	D	P	P	P	P	Reason for Elimination
		U		U		U	D	U	
	1	•	•	•	•	•	•	•	Unallocated cannot be P,D,U
X	2	3	•	•	4	5	•	•	Executable cannot be <i>DIRTY</i>
W	•	•	•	•	•	•	•	•	No Write only pages
W X	•	•	•	•	•	•	•	•	No X and (R or W)
R	6	7	•	•	•	•	•	•	No D and P without W
R W	8	9	10	11	12	13	14	15	
R X	•	•	•	•	•	•	•	•	No X and (R or W)
R W X	•	•	•	•	•	•	•	•	No X and (R or W)

Table 5.1: All Possible MMU Page Conditions

5.2.2 Mapping of Word Use to *FLG*

The MMU has to maintain much permission information and status for each physical page. Here are the conditions (with the *FLG* abbreviations that represent them) that naturally suggested themselves:

R The page is readable by the process.

W The page may be written by the process.

X The page may be executed by the process.

P The *SID* is the default *SID* for this type of page (program or data) for this process.

U This page has been *USED* (executed, read or written).

D This page has been written, *DIRTY*.

If one bit was used to represent each of these conditions, the flag field would require six bits instead of the allotted four.

Many of the combinations do not make sense or represent conditions not allowed by the design considerations of Chapter 3. To see these nonsensical combinations, we constructed a truth table (Table 5.1). A bullet (•) in this table indicates that the entity is not a viable alternative. A number indicates that this entity represents a useful combination of attributes and should be represented in the MMU registers.

With only 15 usable states to represent, only four bits of state will be required. We reorganized the states as shown in Table 5.2. The numbers at the left of the table are the two high-order bits of the flag field in the MMU word. The numbers at the

			<i>USED</i>	Primary	Primary <i>USED</i>
	<i>FLG</i>	00	01	10	11
Executable	00	2	3	4	5
Read Only	01	6	7	1	
Read or Write	10	8	9	12	13
(Read or Write) and Dirty	11	10	11	14	15

Table 5.2: MMU Condition Assignments

top of the table are the low-order bits of the flag field. The numbers inside the table correspond to numbers in the first table.

With this bit assignment, the third bit becomes the *USED* bit, the fourth bit is the Primary bit, and the first and second bit must be taken together to interpret the permissions. The combination 0110 represents an unassigned physical page.

5.3 MMU Operations and Exceptions

The MMU must perform several operations. The operations may not work properly; they may cause exceptions that result in execution of an interrupt handler on the CPU. The operations and related exceptions are described below.

5.3.1 Normal Read and Write

The MMU registers can be read and written by the superuser process, *PID* = 0. The MMU registers are addressed as normal memory, so the MMU must recognize virtual addresses starting at FFFF-800000 and respond to them with the MMU registers rather than trying to calculate a physical address.

Possible exceptions:

- Memory not present
(Addressing MMU register not installed)
- Memory permissions incorrect
(*PID* ≠ 0)
- Flag 0111 not permitted
(Unassigned Flag combination)

5.3.2 From *PID*, *VP* Get *PP* for an Operand and Check Permissions

When presented with a *PID*, a *VP*, and a signal that this fetch is for an operand, the MMU must supply the default OSID, determine the correct *PP*, and check permissions.

Possible exceptions:

- *PID*, *SID*, *VP* not in MMU
- Memory permissions incorrect

5.3.3 From *PID*, *VP* Get *PP* for Text and Check Permissions

When presented with a *PID*, a *VP*, and a signal that this fetch is for an instruction, the MMU must supply the default OSID, determine the correct *PP*, and check permissions.

Possible exceptions:

- *PID*, *SID*, *VP* not in MMU
- Memory permissions incorrect

5.3.4 From *PID*, *OSID*, *VP* Get *PP*, and Check Permissions

When presented with a *PID*, an *SID*, a *VP*, and a signal that this is an operand fetch, the MMU must determine the correct *PP* and check permissions.

Possible exceptions:

- *PID*, *SID*, *VP* not in MMU
- Memory permissions incorrect

5.3.5 From *PID*, *ISID*, *VP* Get *PP*, and Check Permissions

When presented with a *PID*, an *SID*, a *VP*, and a signal that this is an instruction fetch, the MMU must determine the correct *PP* and check permissions.

Possible exceptions:

- *PID*, *SID*, *VP* not in MMU
- Memory permissions incorrect

5.3.6 Change Primary *OSID*

When directed by the CPU, change the primary *OSID* to the *SID* provided on the low-order 16 bits on the data bus. This update requires setting the Primary flag on all entries with the *PID* and new *OSID* and resetting the Primary flag in all MMU registers with the *PID* and the old *OSID*.

Possible exceptions:

- *PID, SID* not in MMU
(An authorized page has been paged out.)
(This *PID* is not authorized to share this page.)
- Memory permissions incorrect
(The new segment identified by *SID* is not writable. The primary operand page must be writable. This is not strictly required, but is the proper thing to do.)

5.3.7 Change Primary *ISID*

Branch instructions may specify addresses in segments other than the primary segment by using appropriate indirect addressing modes. When such a branch is taken, the MMU must update the primary *ISID* for this process. This update requires setting the Primary flag on all entries with the *PID* and new *SID* and resetting the Primary flag in all MMU registers with the *PID* and the old *ISID*. The implementation of the CLOCS architecture must provide communication between the CPU and the MMU to update correctly the *ISID* in this situation.

Possible exceptions: none

The exception *PID, SID, VP* not in MMU cannot occur for this operation, because the MMU must first fetch the new instruction using one of the above operations. If there is an interrupt, the branch instruction will be restarted, so at the time the branch is taken the physical page is available. Furthermore, the instruction fetch operation will verify that this page contains executable code, so no Memory Permissions Incorrect exception may occur.

Some implementations may not specifically use this MMU operation. If the CPU does not fetch the instruction until it decides to take a branch, then the MMU may automatically change the default *ISID* during the fetch. This possibility is left open to the implementers.

5.3.8 Reset *USED* for All Physical Pages

When the operating system selects a page to swap out of main memory it may reference the *USED* bit as described below. Frequently, the operating system will want to set all *USED* bits to zero. To set the *USED* bit for all physical pages to zero, a program writes a word with the low-order bit set to the memory location FFFF-FEFFF8. The MMU intercepts the reference to that special memory location and interprets the low order bit set to one as a command to reset the *USED* status for all installed MMU registers. That location when read contains the number of MMU registers installed.

Possible exceptions: none

CLOCS explicitly supports this operation with hardware because it would be very time consuming for a program to update each individual MMU register. Many algorithms call for updating all *USED* bits each time a page fault occurs, so this clearing operation would be frequent. On the other hand, the *DIRTY* bit only need be reset when a page was written to the backing store. Consequently, it is reset by the operating system writing to the MMU register.

5.4 Implementing Common Virtual-Memory Operations

We decided that the CLOCS architecture should support the three of the most common virtual memory algorithms described in Dietel [9]: write-back virtual memory, copy-on-write, and not-recently-used replacement.

5.4.1 Write-Back Virtual Memory

Before a page may be removed from physical memory, the *DIRTY* status should be checked for any MMU register referring to that physical page. Saving the page on disk before reusing the page is required only when the *DIRTY* status is set. This method significantly reduces memory traffic because much data memory is read, but not changed before it is paged out.

5.4.2 Copy-on-Write

Copy-on-write is an algorithm frequently used by *UNIX* operating systems and *VAX* computers. A process is assigned a block of memory containing information or code. As long as it does not change this memory, it shares the memory with another process. As soon as the process attempts to change the memory, the operating system must intervene to make a separate copy for this process, and then allow the change to happen. This facility is very useful for the "fork" system call in *UNIX*. Copy-on-write may be simulated by assigning the page as a shared, read-only page. In this case *shared* simply means that the page has more than one MMU register pointing to it. When the process tries to write to the *copy on write* page, the MMU causes an exception. The operating system exception handler then copies the page to an unused physical page. It then corrects the MMU register to point to the new physical page and restarts the user process with the instruction that caused the fault.

5.4.3 Not-Used-Recently Page Replacement

One popular page replacement algorithm is not-used-recently. This technique is described in detail in Deitel [9]. Briefly, when a page fault occurs, the operating system selects a page that does not have the *USED* bit set to swap out. It then can use the vacated page to load the needed page that caused the page fault. Deitel points out that a *USED* bit and a *DIRTY* bit must be maintained for each page. CLOCS maintains this information in the MMU.

5.5 The MMU Designs We Discarded

During MMU design, we considered several schemes. Some of the alternate design were interesting to us or involved important design decisions. Some of the designs we threw away are described here.

5.5.1 Alternate A - Virtual and Physical Tables

In the alternate A design, the MMU contained two tables instead of one. One table, Table1, contained a *PID*, a *FLG*, and a *SID* for each entry. The other table, Table2, held a Dirty bit, a *SID* and a *VP* for each entry. Table2 had one entry for each physical memory page, so the *PP* did not have to be included in the table. The advantage

of the alternate A scheme was that it was more proper for support of lightweight processes. The *PID*, *SID*, *FLG* relationship was unique. The chosen scheme was better in that it could support heavyweight as well as lightweight processes and also could resolve the permissions down to the physical-page level. With the chosen scheme, one segment could hold both code and data space on separate pages, so small processes need not take up two segments of address space. The other difference between the schemes was the simplicity of the data structure (only one type of MMU register) and duplication of *PID*'s for the chosen scheme and duplication of *SID*'s in the alternate A scheme.

The final decision of which scheme to use was based on the projected silicon area of the two schemes. We assumed field sizes to be the same for the two schemes except that the alternate A scheme needed one extra *DIRTY* bit. *PID*, *FLG*, *SID*, and Virtual Page fields were all associative memory. The fact that these fields were associative was important because associative bits would require at least 25% more silicon area to implement. Most associative-bit implementations would require about 50% more area than a non-associative bit.

To compare the two schemes, we specified a computer system with 4000 pages of physical memory and capable of running 1000 processes. This machine was a representative system for using the power of the CLOCS architecture and supporting large applications. For a machine of this size, the chosen scheme required 4500 table entries (one for each physical page plus 500 for memory sharing). Each entry was 64 bits long, 44 of which were associative. The Alternate A scheme required Table1 with 2500 entries, two for each processes (one data, one code) and 500 extra for memory sharing. Each entry in this table was 34 bits long, and all were associative. The second table, Table2, contained 4000 entries, one for each physical page. Each entry was 29 bits long, and 28 of them were associative. Table 5.3 shows the bits and the relative area for the two schemes. The column labeled "Total Relative Area" is the total area of the table in non-associative bits, assuming that associative bits are 50% larger than non-associative ones.

The small additional cost of associative bits and the increased function of the chosen scheme, particularly since the chosen scheme supported heavyweight processes (a concept used by many available operating systems) tipped the scales in favor of the primary scheme.

MMU Scheme and Table	Associative Bits	Total Bits	Total Relative Area
Alternate A Table1	80,000	85,000	125,000
Alternate A Table2	112,000	116,000	172,000
Alternate A Total	192,000	201,000	297,000
Chosen Total	198,000	288,000	387,000

Table 5.3: Comparison of Chosen and Alternate A MMU Designs

5.5.2 Alternate B - Some Registers Permanently Mapped

The alternate B MMU design attempted to reduce the number of bits of memory in the MMU and to make some operating-system functions more efficient by permanently assigning some of the MMU registers to physical pages. In this scheme, memory locations FFFF-800000 through FFFF-F00000 were assigned to physical pages zero through 262,144. These memory locations always returned the corresponding physical page number when read, and the physical page was ignored during writes to these MMU registers. In other words, these MMU registers appeared like the other chosen scheme MMU registers except that the *PP* could not be changed. The memory from FFFF-F40000 through FFFF-FEFFFFE could be assigned to any physical page.

The advantages of this alternate B scheme were fewer memory bits in the MMU and a possible improvement in operating-system speed. If a computer system had N physical pages and allowed for an additional M pages to be shared, then $N + M$ MMU registers would be required. We estimated that the alternate B scheme could have saved $N * 18$ bits of memory over the primary scheme. Another advantage for this scheme was improved performance during a naive search for a page to swap out. With alternate B, a search for a potentially shared page would have required only $O(M)$ steps while the chosen scheme takes $O(N + M)$. As estimated above, M would be only 10% of N , so the new scheme would have yielded an order of magnitude performance improvement. This advantage disappeared, though, when Bill Gallmeister suggested a $O(\log M)$ software algorithm. The data structures and algorithm to attain this superior level of performance are well understood[38, 9, 17].

With one major advantage of this scheme eliminated, the disadvantages became more persuasive. Having two classes of MMU registers lacked propriety. Although the same operations could be performed on the two types of MMU registers, different

actions resulted. If the systems programmer made an error and tried to set the physical page number of one of the permanently assigned MMU registers, the action would be ignored and the programmer would receive no warning of his error. An additional disadvantage of the alternate B scheme was that the number of shared pages was limited to M . With the chosen scheme, all MMU's registers may be used for shared pages, with the only disadvantage that some physical pages may not be accessible, a much more graceful degradation of performance.

Since the only advantage to alternate B was the saving of some memory in the MMU and it introduced such serious impropriety, we selected the chosen scheme over it.

Chapter VI

Quantitative Analysis of CLOCS

This research would not have been worth the trouble without some indication that the CLOCS computer would perform reasonably well. When we began the study we estimated that gains in context switch performance could compensate for a throughput performance loss of 75%. That is, if CLOCS took less than four times longer to run a program as a conventional computer architecture, then its improved context-switch performance would compensate, and CLOCS would be the better architecture for enough applications to warrant further study.

To estimate the relative performance of CLOCS to conventional architectures, I performed several analyses that are reported in this chapter. A small exercise checks that the architecture does not have any flaws that prohibit efficient programming. CLOCS performance is estimated using expected bandwidth requirements and a comparison to the MIPS R2000. A final check of the CLOCS architecture's potential is the application of Fuller's evaluation techniques. These analyses all show good potential for CLOCS.

6.1 Programming a Small Problem on CLOCS

In an attempt to reveal major flaws in the CLOCS architecture, I examined a few small problems to identify inconvenient features of the architecture. It was easy to write CLOCS assembler code for small programs involving pointers, subroutine calls and Input/Output.

As an example of these small programs, I present the solution to Exercise 9-2 in the computer architecture book by Blaauw and Brooks[3]. They suggest a character translation exercise to be programmed on different computer architectures. The problem statement is below, and the solution is in Figure 6.1.

9-2 In a stream of 1000 characters of running text, characters are to be replaced according to the following table:

```

; Program Translate
.data
input:                ; Input data
.ascii "Some text of length 1000"
output:               ; Output data
.space 1000          ; Reserve space for 1000 characters
table:                ; 256 character translation table
.ascii "ZBCDEFGHI6KLMNOPQRSTUVWXYZ1294567890:,,:\ " ...

table_pointer:        ; pointer to the translation table
.P table
input_ptr:            ; pointer to input data
.P input
output_ptr:           ; pointer to output data
.P output
count:                ; counter for characters remaining
.di 1000

.text                ; Begin the program.
start:                ; Zap last 8 bits of table pointer.
movqi @input_ptr,table_pointer+7
                    ; Move the selected character.
movqi @table_pointer,@output_ptr
                    ; Increment the pointers.
adddi <1,output_ptr
adddi <1,input_ptr
                    ; Decrement the count and loop.
subdi <1,count
bne start,count
                    ; Return to main program
.end

```

Figure 6.1: Solution to Exercise 9-2 from Blaauw and Brooks

.	:
;	,
3	9
A	Z
J	6
Any invalid character	blank

All valid characters are to remain unaltered. The character set has 48 characters. Write a program, assuming input and output are in memory.

Although CLOCS has no special character handling operations or looping instructions, the availability of all operations for all data type sizes allows a simple inner loop. This program does not reveal any weaknesses in the CLOCS architecture. Note that the loop ending instructions calculate a new value for `count` then use it in the following instruction. This is a common construct in CLOCS assembler code produced both by humans and compilers.

6.2 Expected Memory-Bandwidth Requirements

CLOCS requires higher CPU-to-main-memory bandwidth than many contemporary computer systems because of the flattened memory hierarchy. There are two main causes for this higher bandwidth requirement: all of the data is stored in memory (there are no registers to store intermediate results), and instructions and data may not be cached.

I estimate that CLOCS uses three data references for each instruction. For an instruction that uses direct addresses, the three memory references are:

1. Fetch Operand 1.
2. Fetch Operand 2.
3. Store the result at the address Operand 2.

This estimate of three data memory reference per instruction on average is based on the assumption that the number of instructions requiring less than three references compensates for the instructions requiring more than three. For example, instructions that use immediate data do not have to reference memory to obtain Operand 1, so they use less than three operand memory references. Full-word moves do not have

to fetch Operand 2 from memory, so they also require one less memory reference. The conditional-branch instruction requires only one data-memory reference, and unconditional branches require no data-memory references. On the other hand, an instruction with one indirect addressed operand uses four memory references, and instructions with indirect addresses for both operands use five references to obtain and store the operands. Assuming that the average CLOCS instruction requires three data-memory references and one instruction-memory reference, I estimate that the average instruction will reference memory four times. This estimate is conservative; the Dhrystone benchmark program requires 3.29 references per instruction, and an implementation feature further reduces to below 3.0. See Chapter 8 for the specifics.

Other common computer architectures reference data memory much less frequently, about two memory references per instruction. Most IBM System/360 instructions reference one data memory location; these instructions require two memory references. RISC architectures frequently have separate load and store instructions constituting about 30% of all instructions executed[34, 20]. Therefore, seven instructions operate on the data referenced in three load/stores. To execute 10 instructions, 13 memory references are required: 10 for instruction fetches and three for data fetches. However, only 10 of those instructions really do work (not counting loads and stores as computation), so the memory references per instruction doing real work is $13/7 = 1.85$. These two examples show that conventional and RISC architectures use about two memory references per instruction.

Because CLOCS requires about twice the memory access of RISC architectures, a CLOCS computer system needs twice the bandwidth. Higher bandwidth may be obtained by using faster memory at greater cost or implementation techniques such as dual-ported memory systems. However, even with the best technology available, the CLOCS computer would likely run slower than a contemporary RISC-based computer. The calculations above show that the expected degradation from memory bandwidth is approximate 50%, well about the 25% estimate mentioned in the introduction to this chapter.

Most modern microprocessor-based computer systems use caches to improve performance. In 1989, many microprocessors introduced from major vendors featured cache memory on the microprocessor chip (e.g., the Intel 80860 and 80486 and Motorola 68040). However, using a cache adds hierarchy that violates the major design goal of CLOCS. As a consequence, CLOCS must fetch all instructions and data from memory, rather than from an on-chip cache. To compensate for the lack of cache

Type Instruction	Instruction Frequency	R2000		CLOCS	
		Memory Operations	Weighted Operations	Memory Operations	Weighted Operations
Calculation	30	1	0.30	4	1.20
Branch (Cond)	12	1	0.12	2	0.24
Branch	8	1	0.08	1	0.08
No Operation	20	1	0.20	0	0.00
Load or Store	30	2	0.60	0	0.00
Total			1.30		1.52

Table 6.1: Comparison of CLOCS and R2000 Memory Use

memory, CLOCS must use a higher-performance memory system than contemporary designs or suffer another performance degradation. This degradation is difficult to estimate without implementation details, but I assume it will not reduce CLOCS performance by more than another 50%. Therefore, CLOCS estimated performance is sufficient to warrant more detailed simulations to obtain better estimates of performance.

6.3 Favorable Comparison with MIPS R2000

To investigate the effect of our two-operand, memory-to-memory architecture, we compare CLOCS to a MIPS R2000. The purpose of this comparison is to find an upper bound on the CLOCS performance relative to the R2000, so assumptions are made in favor of better performance for CLOCS. Data from runs of the CAD tool Timberwolf [26] show that the R2000 dynamic-instruction utilization is about 30% loads or stores, 20% branches, 20% no operations, and 30% computation. These are typical values for contemporary RISC processors.

Table 6.1 shows the dynamic-instruction execution percentages along with the number of memory operations required for the R2000 and for CLOCS. By weighting the number of memory operations with the fraction of instructions executed and totaling them, the average number of memory accesses per R2000 instruction is calculated to be 1.30. Similar weightings for CLOCS give 1.52, which represents the average number of memory references to do the same work as the R2000 instructions. The CLOCS value is only 17% higher, showing only a small architectural advantage for the R2000. This difference can be reduced to only 11% using the short-circuit

implementation feature discussed in Chapter 7.

This small comparison does not prove that CLOCS is potentially as fast as a MIPS R2000, but the result does provide some insight concerning the relative performance of the two approaches. As mentioned above, the effect of cache is difficult to estimate without detailed simulation data. Since the R2000 cache runs faster than the CLOCS memory and the R2000 main memory runs slower than CLOCS memory, I assumed no average difference in memory reference time. This assumption also ignores the effect of large numbers of tasks overflowing the R2000 cache, eliminating its advantage. Overall, because the analysis neglects cache and memory-latency effects, CLOCS has probably been given an unwarranted advantage. Another favorable assumption I make is that the R2000 is really not able to take advantage of the data stored in its 32 registers and its three operand instructions, so CLOCS can perform as well with the same number of operations. In spite of the advantages given to CLOCS, the close result gives hope that CLOCS throughput performance will not be much worse than the R2000, and certainly will be within the factor of four mentioned above.

6.4 Using the "Measurement and Evaluation of Alternative Computer Architectures" Technique

In the mid 1970's, Samuel H. Fuller, while working with the Army/Navy Computer Family Architecture (CFA) Committee, developed an approach for quantifying the relative performance of alternative computer architectures. The committee's methodology and their evaluation of nine computer architectures was published in 1977[16].

The criteria the CFA committee used placed great importance on servicing interrupts and exceptions. The analysis method used memory transactions measured in bits to assess costs of these operations, and they used the resulting cost as part of a final weighting of each architecture's merit. This approach has been criticized by advocates of RISC architectures because it did not directly measure throughput performance for calculations and consequently rates modern RISC machines poorly[20]. The analysis did not directly measure the memory bandwidth required for typical calculations. As a result, performance improvements from the use of cache memory and large numbers of registers do not help an architecture's score for this analysis. Instead, since the registers would have to be saved at context-switch time, the large number of registers in a typical RISC design would unfavorably affect the context-

switch measures and ultimately degrade the final score. However, the CFA committee was looking for computers for *ruggedized* military use. By *ruggedized*, they implied computers for embedded applications such as combat information computers on ships or navigation and target tracking computers on aircraft. Most of the applications would involve real-time or near-real-time tasks. The criteria were based on their expected applications.

The sections that follow explain the criteria that Fuller defined and explain how I assigned values for CLOCS. The last section reports the results a Fuller-type analysis of CLOCS and the machines considered by Fuller.

6.4.1 Fuller's Criteria

Fuller specified both absolute and quantitative criteria. The absolute criteria had to be satisfied for the architecture to be acceptable for further evaluation. The quantitative criteria were combined to give a relative rating. In his study, the three architectures with the highest composite score were further studied by programming and running several sample problems.

Candidate architects had to meet *nine absolute criteria*. Apparently some of the criteria were sufficiently vague that the CFA committee disagreed on whether machines met them. In the published report, two cases were unresolved, and several were marked as questionable. The criteria are listed below, some with brief definitions.

1. Virtual memory support — address translation
2. Protection — experimental applications do not endanger other programs
3. Floating-point support
4. Interrupts and traps
5. Subsetability — working computers may be built with certain subsets of features
6. Multiprocessor support — a test and set instruction or equivalent
7. Controllability of I/O — a criterion vague enough that all architectures appear to satisfy it
8. Extensibility — at least one spare operation code

9. Read-only code — programs must be able to reside in read-only memory while they are executed

CLOCS meets all of these criteria. Of the nine architectures that Fuller investigated, only two met all the criteria without question. However, most new microprocessor designs meet all of the criteria, reflecting progress in the field that has been incorporated in the CLOCS architecture.

Fuller specified 17 quantitative criteria. Groups of criteria are discussed in paragraphs below:

Four criteria concern address-space sizes. *V1* is the number of bits that can be addressed using virtual addresses. *V2* is the number of bits that can actually be accessed using virtual addresses. For CLOCS, both values are 2^{57} : 14 bits from the PID, 16 from the SID, 24 from the offset, and three for the eight bits per addressed byte. Similarly, *P1* and *P2* concern the physical memory. Both *P1* and *P2* are 2^{35} : 32 for the physical address and three for the eight bits per address.

U is the fraction of the operation unassigned codes; it is $24/32 = 0.25$ for CLOCS.

CPU state size is measured by four criteria. *CS1* and *CS2* are the size of state for full and subsetted architectures. Both values are 64 bits for CLOCS. *CM1* and *CM2* are the number of bits transferred between the CPU and main memory on a context switch for full and subsetted architectures. This value is 128 for CLOCS, since the machine automatically writes the old status word to memory and fetches a new one upon interrupt or trap.

K is assigned the value one for computer architectures that may be virtualized, zero otherwise. CLOCS meets this criterion.

B1 and *B2* are the usage base prior to June 1, 1976. CLOCS has no delivered computers.

The number of bits of memory traffic to start an input/output(I/O) operation is *I*. From the numbers assigned for the other computer architectures, I inferred that no error checking is required. Therefore, to start an I/O operation, CLOCS uses 128 bits: one 64-bit word is fetched to get the instruction, then one 64-bit word is written to the memory address associated with the I/O device.

D is the number of bits that each instruction can directly address using only one base register. For CLOCS, this is 2^{27} , representing 24 bits of address offset and three bits for the eight bit byte.

L is the interrupt latency, expressed in bits transferred between memory and

processor. This is the same as a context switch for CLOCS (128 bits).

Subroutine linkage efficiency is measured by $J1$ and $J2$, the number of bits required for a subroutine call with no parameters with and without floating point, respectively.

6.4.2 CLOCS and Other Architectures

To obtain a composite quantitative score, the CFA committee assigned weights to each category. To calculate final quantitative scores for each architecture, all values were normalized. Criteria concerning address space were further adjusted to obtain a standard deviation of 1.00. The CFA committee added this normalization of the standard deviation for address space measures because they did not feel that several orders of magnitude of address space represented several orders of magnitude of merit. The resulting total scores had an average of 1.00.

I added the values for CLOCS to this calculation. This extra architecture required recalculating all the averages and standard deviations for each of the criteria. This recalculation changed the absolute values, but not the relative order of the composite scores. Table 6.2 presents the criteria values and final composite scores for CLOCS and the top three scoring original architectures. The Interdata 8/32 had the highest original score, followed by the IBM S/370 and the PDP 11. CLOCS scores considerably better than the other architectures because of significantly better values for address space size and context-switch metrics. The advantage in these areas was partially offset by the large word size since all of these metrics were based on the number of bits transferred between the CPU and main memory. The only areas where CLOCS scored poorly were I/O and installed base. The I/O score was not too bad, and the weights for installed base were relatively low.

All in all, this analysis shows CLOCS to be a good potential architecture for military computers.

6.5 Summary

The quantitative analyses of CLOCS show good potential. The machine is not difficult to program. Memory utilization is high, but not worse than about two times that of other machines. The simple comparison of CLOCS with the R2000 running Timberwolf shows that CLOCS may perform competitively under some conditions. Finally, a detailed evaluation method (the Fuller-type analysis) shows CLOCS to

Quantitative Criteria	Criteria Weights	IBM 370	Interdata 8/32	DEC PDP-11	CLOCS
V1**	0.0433	27	27	20	57
V2**	0.0529	27	27	19	57
P1**	0.0612	27	27	25	35
P2**	0.0554	27	27	24	35
U	0.060	0.371	0.355	0.043	0.250
CS1	0.0466	1344	1632	1168	64
CS2	0.0371	576	576	144	64
CM1	0.0596	3168	1120	736	128
CM2	0.0450	1312	32	480	128
K	0.0558	1	0	1	1
B1	0.0313	17300	185	14700	0
B2	0.0254	16000	14	311	0
I	0.1238	64	16	16	128
D**	0.1025	15	27	19	27
L	0.0917	6192	560	112	128
J1	0.0629	1904	2368	1040	1024
J2	0.0475	1136	1280	400	1024
New Score		1.00	1.30	1.28	2.39
Old Score		1.36	1.68	1.43	

** These values are of the form 2^X where X is the indicated data.

Note: Data for the ROLM AN/UYK-28, UNIVAC AN/UYK-7, SEL 32, Burroughs B6700, UNIVAC AN/UYK-20 and Litton AN/GYK-12 were used for the computation but the results are not shown here to conserve space.

Table 6.2: Results of a Fuller-Type Analysis

have excellent potential for applications such as those run on military computers. The next chapter addresses implementation designs to assess whether a CLOCS computer could be feasibly built.

Chapter VII

Implementation of the CLOCS Architecture

During my research, I investigated several different implementations of the CLOCS architecture. This chapter begins by differentiating architecture and implementation and presenting common implementation techniques. As with Chapter 5 on the MMU, this chapter will describe the chosen implementation design, and some of the discarded designs.

I attempted to design implementations reasonable for a microprocessor realized in a technology that could place one million transistors of random logic on a single chip. The major assumption was that the CPU could accomplish just about anything in the time required for a memory operation. This assumption is not completely valid; for example, a floating-point multiply in one cycle is quite challenging. However, for the purposes of this research, these assumptions will give sufficiently accurate timing results, because long operations are infrequent.

This chapter begins with an explanation of the purpose for spending the research time to create implementations. A tutorial on Central Processing Unit (CPU) implementations follows. The next section presents the chosen implementation and defines the operations added to ease the task of implementation and simulation. The remainder of the chapter deals with discarded implementation designs.

7.1 Why Implement CLOCS?

Since we are dealing with computer architecture research, one might ask “Why implement the CLOCS machine?” (By *implement* I mean *design a computer system in enough detail to estimate the time required to execute a program.*) A radically different architecture must perform well in the real world in order to be useful. The architecture does not specify any timing information, so a more detailed definition of the machine is required that allows reasonably accurate estimates of program-execution performance.

An *implementation* is a specification of a computer at a lower level than architecture, but at a higher level than a realization. Architectures give no timing information at all: they simply describe the way the machine will function, i.e. what programs will run and what answers they will produce. An architectural diagram consists of functional units selected for ease of understanding and with no regard for the actual possibility building a system. An implementation breaks the functions up enough that reasonable relative-timing estimates may be made for each part. The realization will specify the physical layout and the types of the parts well enough for precise timing to be determined. Blaauw defines these three levels in [2] thus:

The design of digital systems can be viewed from three levels. The highest level concerns the architecture, which specifies the functional behavior of a system. The lowest level concerns the realization, and deals with the components from which a system may be constructed. The middle level of systems design concerns the implementation, or the logic structure that embodies the architecture and utilizes the logic of the components of the realization.

In a sense, even the highest level architectural (or functional) simulator can represent an implementation. However, such an implementation may not be economical or buildable. If an implementation requires the memory system to complete three reads and one write each cycle, the realization (the actual building of the system) will be very expensive and probably very slow. Multiported memory systems with four or more ports have been implemented, particularly to support multiprocessor, mainframe-class machines[1], but memory system of this kind would not be appropriate for a single microprocessor. When more than two memory ports are used, handling collisions efficiently becomes much more complicated, and I estimate that it would add too much cost to a CLOCS system to justify the anticipated performance improvement.

The implementation of CLOCS must represent a machine that can be built at reasonable cost. The desired implementation must strike a balance between economy of potential realizations and performance. We want to compare CLOCS to real-world, successful, commercial computers that are realized from an implementation resulting from such a balance.

7.2 What Does an Implementation Look Like?

To reduce the number of objects that the designer must handle, the most basic logic components are grouped to form functional units. The designer may then connect (describe the interaction among) instances of these units to perform the functions required by the architecture description. Since the functional units are formed from basic logic components, the implementation describes how a machine may be realized (built).

What are common implementation functional units? The functional units have well defined inputs and outputs. The functions they perform are easily understood or are commonly required, and the time required for each of them to complete its function is easy to determine. In summary, the blocks are big enough that the designer may describe the machine using them but small enough to permit easy timing estimates and realization.

Common functional units are:

- Registers
- Register files
- Read-only memories
- Small read/write memories
- Adders
- Multipliers
- Data path for arithmetic/logic units
- Finite-state machines
- Combinational logic

These devices can be realized in several different ways. Also, some of the devices can be composed of others. For example, an adder may be made up of pieces of combinational logic. Also, the designer may invent new functional units comprised of existing ones to minimize the number of objects he must handle.

7.3 What Implementation Techniques May Be Used?

There are some very well understood and frequently used techniques of implementation. These ideas certainly help the implementation of CLOCS.

7.3.1 Pipelining

Pipelining describes a type of concurrency with serial restrictions. In this case, it means that more than one instruction or operation is in progress. The different operations must be at different stages of completion. For example, a CPU may have a four-stage pipeline:

Stage	Function
First	Decode Instruction $i + 3$.
Second	Fetch Operands for Instruction $i + 2$.
Third	Calculate Result for Instruction i .
Fourth	Store Result for Instruction i .

Pipelining is a common way of obtaining parallelism in computing because it has fewer problems than other types of concurrency. One disadvantage of pipelining is that the system will not attain theoretical performance because unpredictable discontinuities in the instruction sequence will require the pipeline to be (at least partially) refilled. The most common form of discontinuity of the instruction sequence is a conditional branch, which occurs approximately every eighth instruction during the execution of most programs. If the pipeline does not handle unconditional branches efficiently, the discontinuities occur more frequently, since branches make up 20% of all instructions[26, 34, 20]. Another source of discontinuity is a context switch, but that occurs at a considerably lower rate for most applications. More importantly, to execute programs correctly this technique must work around two significant difficulties: control of the pipeline and data dependencies.

First, advancement of instructions through the pipeline must be appropriately controlled. If all stages can always complete their tasks in a constant amount of time, then the pipeline may be advanced at that rate. However, if the amount of time a stage needs to complete its work depends on the instruction it is processing, communications will be necessary to determine when to move the instructions down the pipeline. Providing the interlocks to control the pipeline in this circumstance may add much complexity to the CPU and slow down execution.

The second problem is caused by data dependencies. In the pipeline hypothesized above, consider what happens if the first instruction calculates a value, and the second instruction then uses it? The second instruction will try to fetch the value at the same time as it is being calculated, and the first instruction will not get around to storing it until later. The data-dependence problem is sometimes made a visible feature of the architecture; in such a case it is up to the programmer or the compiler to avoid data dependency problems. Implementations of architectures that conceal the problem commonly use one of two solutions: scoreboarding (make the fetch stall until the data is written) and shortstopping (taking the answer directly from the calculation step to the fetch step). Note that scoreboarding leads to pipeline interlock problems, and shortstopping adds considerable complexity to the CPU[36, 3, 31].

In spite of the difficulties, pipelining is effective. If the architecture avoids or minimizes the difficulties, a good four-stage pipeline can approach a four-fold increase in peak throughput performance. Most RISC computer designers carefully considered pipelining as they selected features for their architectures; perhaps the greatest contribution of the RISC design methodology was easing of pipelining.

7.3.2 Data Caching

Frequently a program will access the same piece of data more than once, with very little intervening time, as in $x \leftarrow x^2$. The data dependencies discussed in the pipeline section above occur often. If this frequently used data happens to be inside the CPU when it is needed, delays from accessing the memory system may be avoided.

Keeping data in the CPU is a technique dating back to von Neumann[4]. Registers are the most frequently used example of that technique. Unfortunately, the use of registers is the main architectural feature that CLOCS eliminates.

Other related caching techniques are possible, though. For example, several data values could be stored in the CPU as they are calculated. When the value is needed, a memory reference would be avoided. The space of temporary data could be managed using approximations to *Least-Recently-Used* discarding algorithms that are used in virtual-memory systems. Unfortunately, this approach would add to complexity and increase CPU size. Also important, the running time of a program depends on the state of this temporary data area. Accurately predicting the time for a program to run is important to many real-time applications, so they would not be able to use the improved performance from this technique.

It may be acceptable to use data that is already in the CPU. Shortstopping solves the data-dependency problem and eliminates a memory reference. Since shortstopping normally refers to using the data as it comes out of the execution unit, I use the term *short-circuiting* to describe the very similar process of using data in a CPU register to avoid a memory reference. This technique may be successfully used in implementations of CLOCS if it can be accomplished without too much added complexity and if there is little variability of program performance. For example, if only the output of the Alu is used for short-circuiting, only the result from one instruction is lost when a program interruption occurs. The difference in performance is a maximum of one memory reference for each interruption, so performance differs greatly only if a program is interrupted after every instruction (which is an unlikely event).

7.3.3 Multiported Memory

One way for an implementation to improve performance is to allow more than one access to memory at a time. For example, two operands may be needed. The implementation could include two memory-address busses and two memory-data busses. Several designs exist to allow simultaneous access to different memory locations. Dual-ported memory chips exist. Also, the memory system may be made up of several banks of memory. As long as the accesses go to different banks, such a system can easily handle two simultaneous accesses. The more banks, the lower the chances of the same bank is being accessed by both busses. Very little added complexity is required to handle the case of both busses, accessing the same bank, so an interleaved, two-ported memory system may be a very economical performance-improvement technique.

7.3.4 Posted Write

One technique of improving memory system performance is *posted write*[22]. With this technique, the memory system is responsible for completing a write operation; the CPU assumes that the write has completed correctly and immediately continues with the next instruction. This technique is very effective in systems with latency in the memory system, because completing the write does not interfere with instruction execution. However, handling exceptions can be quite complicated, because if the write fails some time after an instruction completes, the effects of subsequent instructions may have to be undone. However, if it does not take too long to iden-

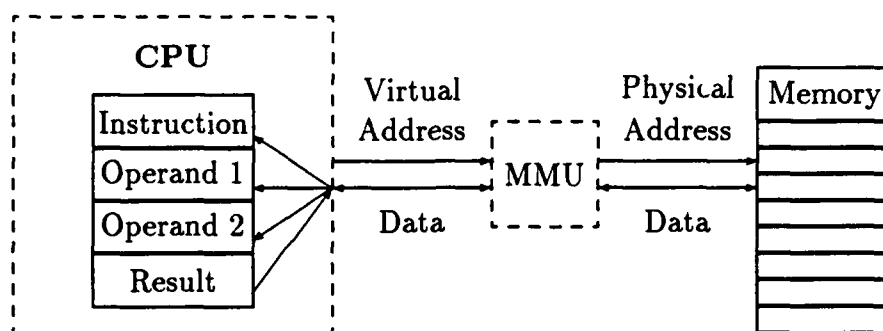


Figure 7.1: CLOCS Implementation Overview

tify exceptions or the CPU does not perform many operations that must be undone, posted write is an inexpensive way to make the memory appear much faster to the CPU than it really is.

7.4 Chosen Implementation

The chosen implementation used to produce the reported simulation results reflects the lessons learned from the previous attempts (discussed later in this chapter). The chosen implementation resembles a hard-wired CISC (Complex Instruction Set Computer) more than a pipelined RISC. The CPU processes only one instruction at a time. Combinational logic units perform each minor required function and the units communicate asynchronously. The state of the instruction resides in four sets of registers that are connected by the logic units. The assumption that activities inside the CPU are relatively fast compared to memory operations justifies our expectation that this implementation accurately estimates the performance of a buildable system.

7.4.1 Implementation Overview

Figure 7.1 is a top-level diagram of the chosen implementation. Four sets of registers are in the CPU, and they contain the instruction, the two operands, and the result of the operation. The CPU communicates through the register sets with the MMU, and the MMU in turn communicates with the memory system.

Each register set in the CPU contains data, but the address associated with the data. For example, the address part of the instruction register is actually the status word (which includes the program counter). Also in each register set are the MMU command and the status of the last MMU command.

The registers are connected by functional units composed of combinational logic.

These units read fields in the instruction and the MMU status, and they modify the MMU command and data fields. These functional units also communicate directly with each other. Because the functional units are stateless and communications are asynchronous, several functions might be accomplished in a single cycle. For example, an immediate value can be assigned to a full word in one clock cycle. The Operand 1 immediate conversion, Operand 2 address calculation, and Alu action (a simple copy in this case) all follow rapidly from instruction decode because no memory access is required.

The registers sets are

- Status word (with program counter) and instruction
- Operand 1 address and operand
- Operand 2 address and operand
- Alu result address and result

The address registers are larger than an address, because they hold other information. This information includes all of the data normally in the status word, such as the Process Identification (PID), and flags identifying the type and validity of the address.

These registers are read and written by nine functional units:

Direct1 Get Operand 1 immediate, or first memory fetch.

Direct2 Get Operand 2, first memory fetch.

Indirect1 Get Operand 1, second memory fetch.

Indirect2 Get Operand 2, second memory fetch.

Alu Calculate results and write to memory.

Check Check for exceptions and cause interrupts.

Mmu Handle memory and memory management operations.

Short-circuit1 Short-circuit Operand 1 memory fetches.

Short-circuit2 Short-circuit Operand 2 memory fetches.

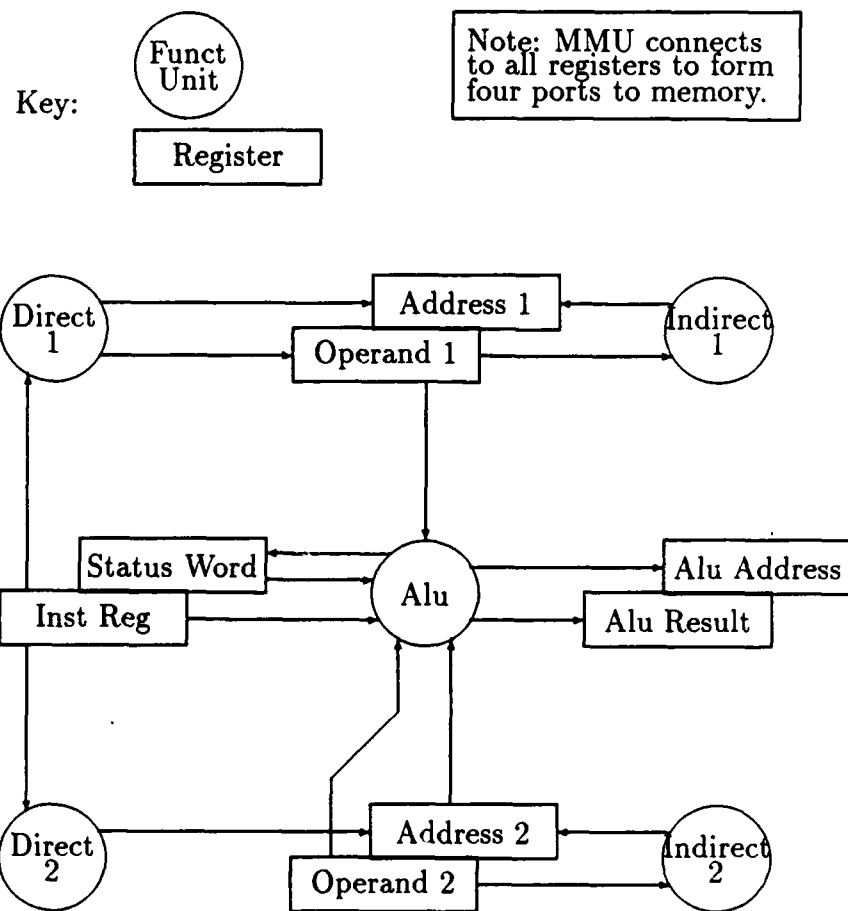


Figure 7.2: Operand and Alu Portion of Chosen Implementation

These functional units (including *Alu* and *mmu*) do not have any state. The terms ALU and MMU refer to higher level components which in some implementations have state. For example, the MMU contains the MMU registers.

The interrelationship between the registers and functional units for the CPU side of the registers is shown by Figure 7.2. In this figure, the register sets are shown as two rectangles: the address and the data. The registers both contain data visible to the architecture, such as operands and addresses, and they also hold the status and commands for the MMU. The memory and exception handling portions of this implementation are shown in Figure 7.3. The same registers appear, but the figure shows their relationship with the MMU, memory, and short-circuit functional units.

Each of the short-circuit functional units continuously compares the address for their operand to the address for the ALU result. If during operand fetching the addresses are the same, the short-circuit unit transfers the data to the operand register. It also cancels the MMU command, which is some form of read operation, and indicates in the operand status field that the MMU has successfully fetched the data. This method of short-circuiting does not require any special logic in the get-operand units (*Direct1*, *Direct2*, *Indirect1*, and *Indirect2*).

7.4.2 Practicality of this Implementation

This implementation is somewhat more aggressive than the Two-Instruction Pipeline model. For the *move* instruction example mentioned above, about 15 gate delays would be required per clock cycle. Given the assumed realization technology, this would be attainable and would not seriously imbalance the CPU and memory system speeds.

7.4.3 Performance Assessment

This implementation attains an average of 2.7 cycles per instruction, a reasonable number for CLOCS. This instruction issue rate uses every memory cycle and is able to avoid another 20% of potential memory accesses through short-circuiting. The implementation is robust and provides for the addition of features. For example, experimental addressing modes and multiport memory were easily added to this implementation.

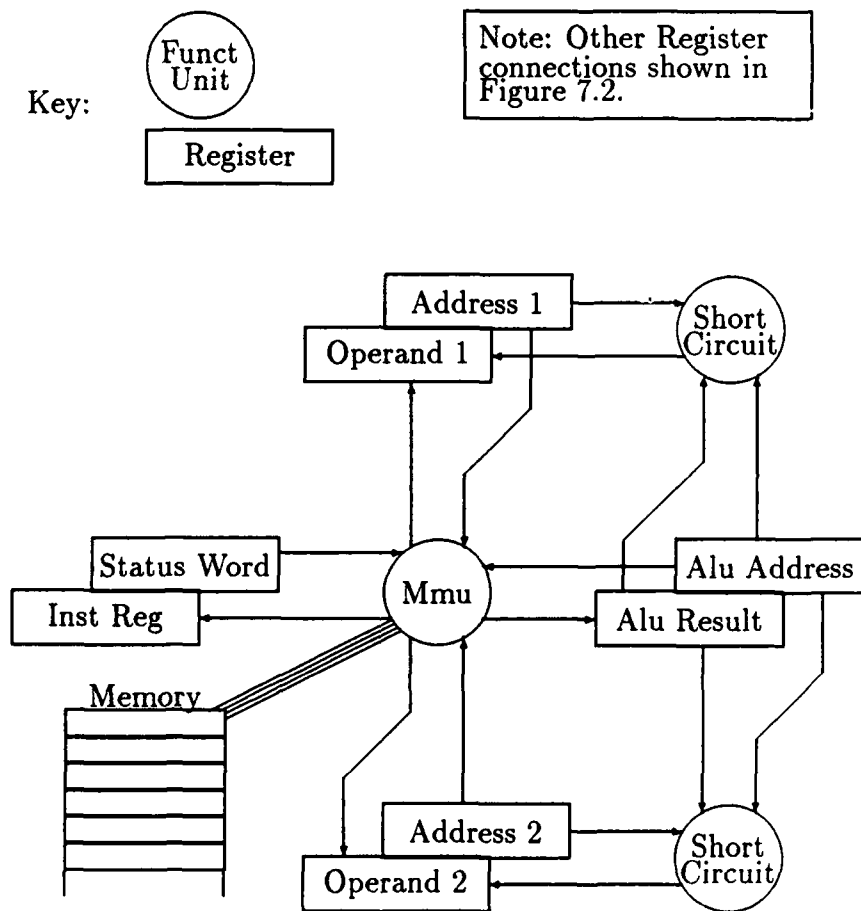


Figure 7.3: Memory and Check Portion of Chosen Implementation

7.4.4 Implementation Feature Additions

The compiler that was used as part of the simulation system (see Chapter 8) assumed the availability of certain operations that were not likely to be supported by most implementations or were not part of the CLOCS architecture. The general design philosophy had been that such cases would be handled by operating-system routines. For example, operation codes were reserved for floating-point arithmetic. If the CPU did not support these operations, it would generate an invalid instruction interruption, which the operating system could catch. The operating system then could call a subroutine to emulate the unimplemented instruction. This approach was justified because typical programs infrequently used these operations. For example, no floating-point instructions appear in the timed portion of the Dhrystone benchmark program. However, writing and debugging the operating-system handling and emulation routines would not be an easy task. The value of producing these routines would be very low to this research effort because these operations are relatively unusual and the technique of emulating unimplemented instructions is well understood[3]. Instead, I simulated all operations of the architecture and added some new ones using spare operations codes.

Floating-Point Operations

Although the important benchmark programs are integer programs, they call system subroutines that contain floating-point operations. Rather than attempt to port IEEE floating-point routines to CLOCS, I just support all of the floating-point instructions in the simulator. I consider implementing floating-point arithmetic in one memory cycle to be unreasonable, especially for divide, but my simulator does this. This unreasonable feature does not affect the benchmark performance, though. It only simplifies the research effort.

Conversion Operations

The GNU C compiler expected size conversion instructions and instructions to convert between integer and floating point. Size conversions for integer numbers were easily emulated in CLOCS assembler (and will affect final benchmark performance). However, size conversions for IEEE floating point are more complicated, as are conversions between IEEE floating point and fixed point. Since these operations would not affect benchmark performance, and were infrequently used, spare opcodes were

assigned and the simulator modified to handle the new instructions. This saved the time otherwise necessary to write subroutines in CLOCS assembler to accomplish these functions and to modify the compiler to call the subroutines.

7.5 Discarded Designs

As soon as architecture definition neared completion, I considered several implementation designs. These implementations were educational, but did not satisfy the requirements. They were discarded because they did not provide timing estimates for systems that could be built economically. Either the individual functional units were too complicated and expensive, or the designs were too wasteful of system resources (some expensive functional units were underutilized). One design, the *Two-Instruction Pipeline*, was more promising, so a simulator was created for this implementation design; however, it too was discarded because it was too inefficient.

7.5.1 Functional

The first and most obvious design was a functional (or architectural) simulator. Part of an architectural simulator was actually built to support C language compiler work. It only simulated CPU operation. A complete simulator, able to run the designated benchmark programs, could have been provided by adding memory management and instrumenting the design. Such a simulator could have provided data on the number of instructions executed. The biggest flaw with this approach is the implicit assumption that processing each instruction is the most important and time-consuming task, and that counting the number of instructions executed accurately estimates the time required to run the program. This approach explicitly assumes that the number of memory accesses does not affect the running time of a program, clearly a false assumption. In reality, the memory accesses drive performance on this memory-to-memory machine. Even with further additions to the simulator to count memory accesses, this enhanced simulator could not be used to investigate realistic memory systems or describe a system that could actually be built. As a result, the functional design was scrapped in favor of an implementation centered around memory access.

7.5.2 Seven-Instruction Pipeline

To more effectively attack the memory-system performance problems, the next implementation design isolated each type of memory access. An instruction pipeline would hold different instructions, and one type of memory operation (or ALU operation) would be accomplished at each stage.

The pipeline stages were:

Stage	Function
First	Instruction Fetch for instruction $i + 6$.
Second	Operand One Address Fetch for instruction $i + 5$.
Third	Operand One Data Fetch for instruction $i + 4$.
Fourth	Operand Two Address Fetch for instruction $i + 3$.
Fifth	Operand Two Data Fetch for instruction $i + 2$.
Sixth	ALU Operation for instruction $i + 1$.
Seventh	ALU Result Operand Store for instruction i .

In the second and fourth stages addresses were fetched during indirect addressing. Ideally, there would have been work to do in each of the seven stages. However, many CLOCS instructions do not use indirect addressing for both operands, and some use immediate operands or only one operand. As a result, in several stages there was nothing to do.

The idea of this design was that the Memory Management Unit would have six ports, one for each of the pipeline stages that access memory. The MMU would then service all of the ports that had active requests. If the MMU could not perform all required operations, it would stall the pipeline and take any required additional cycles to satisfy the remaining memory-operation requests.

This design better supported memory as the limiting component. Also, because it had a stage for each function and accessing memory once would complete each stage's work, no interstage interlocks were required. Note that I again assumed that the ALU could complete its work in one memory cycle.

The design had three major deficiencies. First, this design did not benefit much from the very long pipeline because of frequent flush and refill of the pipeline. Branches could be expected every four or five instructions, and, because the CLOCS architecture contains no special operation code for unconditional branch, even unconditional branches would cause some pipeline difficulties. As a result, keeping the pipeline full would have been difficult. The expense of all the pipeline stages would

have been hard to justify, since, for a typical set of seven instructions, many of the stages were idle. For example, if the first operand was an immediate constant, the second and third stages had no work to do. The third problem was the worst: the data dependencies caused serious problems in this implementation. If, for example, the pipeline stage that was processing instruction $i + 6$ required a data address calculated by instruction $i + 5$, the result would not be available for four more advances of the pipeline. Although techniques exist to solve these problems, in general those solutions violate the premises of the CLOCS architecture and are extremely expensive. For example, CRISP has a deep instruction pipeline and stores decoded instructions. However, when control transfers to another section of code as the result of an unconditional branch, significant delays occur as the pipeline is refilled[10].

Although this implementation scheme was discarded because the pipeline was too long, instruction pipelining has been a very successful technique for other computer architectures. Consequently, that technique could not be totally ignored, and the next design was also pipelined, but with fewer stages.

7.5.3 Two-Instruction Pipeline Overview

Since memory accesses were expected to limit CLOCS performance, I wanted to ensure that there was always memory traffic waiting. One way to make this more likely was to have two instructions in the CPU, both of which were being processed continuously. I called this the *Two-Instruction Pipeline Implementation*. An implementation level simulator was written, and small test programs were run on this simulator.

I broke up the tasks for instruction processing and assigned them to finite-state machines. The two instructions in the CPU were labeled a and b . Stored with each instruction were the status word (which includes the program counter) and each of the two operands and the address for each of the two operands. I referred to the combination of status word, instruction, operands, and addresses as a line. This was a two-stage pipeline, so there were two lines: a line and b line.

The finite-state machines were autonomous and communicated via clocked control signals. The machines and their functions were:

Nextpc Calculate the next instruction address.

Geti Fetch the next instruction.

Opnd1 Get Operand one.

Opnd2 Get Operand two.

Alu Calculate the result, including branches and store results.

Mmu simulate MMU function

In this design the Memory Management Unit had four ports, and these ports connected to the finite-state machines of the same name in the CPU:

Geti Fetching instructions

Opnd1 Fetching Operand 1 or its address

Opnd2 Fetching Operand 2 or its address

Alu Storing calculated results

The Memory Management Unit serviced the ports in the order listed above. It completed only one memory operation per cycle.

Nextpc calculated the program counter, then activated *Geti* to fetch the instruction. *Geti* then signaled the operand-fetching units (*Opnd1* and *Opnd2*)

Opnd1 and *Opnd2* filled in the operand registers in the *a* line. During an indirect-addressing-mode memory reference, the operand-fetching unit deposited the address temporarily in the operand register while it conducted the second memory fetch to get the actual operand. Before attempting to get an operand from memory, these functional units checked the operand address for each operand in the *b* line. If either of the *b* line operands had the same address, the fetching unit copied the corresponding data instead of generating a memory reference. One additional complication arose if the *b* line Operand 2 contained the desired data. In that case, the operand-fetching unit had to check that the *Alu* had completed its calculations, and that the correct data was actually in the register. Using the data already in the CPU instead of fetching it from memory was called *Short-Circuiting* since it short-circuited the memory-fetch requirements. I suspected the memory savings was valuable, because many times a calculated result would be used in the next instruction, and occasionally the same value for Operand 1 would be reused.

The *Alu* Unit performed the operation specified by the instruction in the *b* line instruction register, using the *b* line operands as inputs, placing the result in the *b* line Operand 2 register, and requesting the Memory Management Unit to write out the data if required.

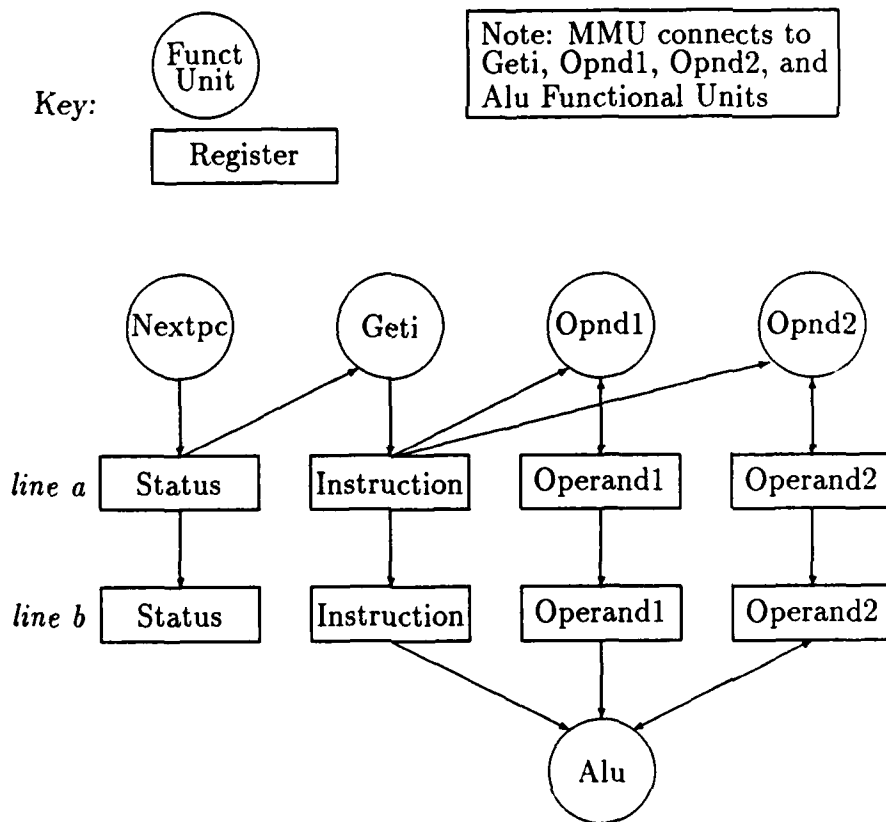


Figure 7.4: Two-Instruction Pipeline Implementation

Figure 7.4 shows the organization of the two-stage pipeline implementation. The memory system is not shown in this diagram because of the numerous connections. With short-circuiting and the MMU, there were eight additional connections.

This design was a conservative implementation. The utilization of the memory management ports was reasonable, and the realization of each of the finite-state machines would have been easy.

7.5.4 Simulation Results and Conclusions

When the simulation program was written, I ran small test programs through it. The simulation had a clock that ran at the memory system access rate; one memory access could be completed in one clock cycle. I was appalled at the poor performance. Instructions' execution times averaged over ten clock cycles. The poor performance was the result of interlock propagation delays. Because the units were simulated as state machines, communications had to wait for clock boundaries. Consider the example of a fetch of Operand 1. The MMU fetched the operand and signaled that the data was ready. Then the Operand 1 Unit signaled to the Geti Unit that the operand was now ready. Then the Geti Unit signaled the Alu Unit to begin work. As a result, the data sat around for two extra clock cycles without any work accomplished on it. To improve performance, a different control scheme would have been required.

Short-circuiting was frequently used for Operand 2; the value calculated in one instruction was used in the next instruction. On the other hand, none of the programs I simulated had instructions that used Operand 1 from the previous instruction. From this, I concluded that Short-circuiting was useful only for Operand 2, the result of the previous instructions calculations.

Chapter VIII

Simulation Programs and Results

To find out how an implementation performs, we use a set of programs that make up a simulation system. These programs compile C language programs into CLOCS assembly language, assemble programs into CLOCS object modules, and simulate running the modules on an implementation. Other parts of the simulation system are scaffolding programs such as debuggers, and library routines for common tasks such as printing. This simulation system is described in the following section. To obtain interesting results from the simulations, we need some test programs. Short programs are interesting because they can be easily understood, but a better indicator of throughput performance is the Dhrystones benchmark program. The second section of this chapter describes these benchmark programs. The third section analyses the results of the simulations. The final section uses the results to compare a 16Mhz CLOCS computer to a DECStation 3100, and from that characterizes applications better suited to CLOCS.

8.1 Simulation System Description

All of the simulation programs are written on a Sun 3/60 running SunOS. Programs are written in C language, Flex (a Lex-compatible regular-expression language), or AWK. These programs are described in more detail in Appendix A, but following are brief descriptions of the compiler, the assembler, the simulator, scaffolding, and library routines.

8.1.1 GNU C Compiler

Version 1.35 of the GNU C language compiler has been ported to generate code for CLOCS. The GNU C Compiler(GCC) is a modern and flexible compiler that performs several different types of optimization. I selected it because of its high reputation and

the fact that many people within the research community were working to improve it. The compiler is available free from the Free Software Foundation and may be freely distributed.

The major part of the compiler unique to CLOCS is contained in the files "md" and "tm.h." The "md" file is a machine description written in a Lisp-like notation describing the architecture. Each entry describes an operation and the operand specifications, and the entry includes a template for the corresponding assembler instruction. Functions not supported by the target architecture may be simulated by entries defining expansions into two or more operations that are supported. For example, CLOCS does not have a compare instruction. Instead, compare is expanded into move and subtract instructions. Peephole optimization may also be specified to combine multiple operations into another that is supported by the architecture. Although the description language is powerful, it is aided by the ability to use C language subroutines in the file "aux-output.c." For CLOCS, this file is only 48 lines, but it contains a subroutine that greatly simplifies the "md" file. The "tm.h" file is a C language header file containing machine-dependent macro definitions. This file describes the machine's registers and provides C language code for tasks such as checking address validity and producing assembly language for function prologues and epilogues.

Other than the configuration files described above, other portions of the compiler required modification. The support for 64-bit integers is not always correct. The compiler also makes some incorrect assumptions about the addressing capabilities of the target machine. Also, version 1.35 of GCC has several unnecessary restrictions concerning data-object sizes. For example, the type *short int* must be one-half the word size. I provided the corrections for these problems to the maintainers of GCC, and version 1.36 of the compiler includes the corrections.

Another facet of the compiler work is noteworthy. The vast majority of data references are to dynamically assigned storage locations on the stack. In order to address those locations, the programs require storage locations whose address is known at compile time. I solve this need by reserving about 64 words in the data space for address and scratch space calculations. I then specify to the compiler that these are registers and let the GCC register allocator manage them. Some locations are reserved for parameters and system pointers. Unfortunately, because C is a recursive language, these registers (really fixed memory locations) cannot be allocated globally to avoid saving them on subroutine calls. Instead, the prologue for each

function copies the contents of fixed locations that are needed onto the stack. At return, these values are restored so the calling program will execute properly. This adds significant overhead to procedure calls. For hand-optimized, assembler-language subroutines that do not call other subroutines, such as "strcpy," this extra overhead may be avoided. Adding optimization to GCC to use fixed-storage locations more efficiently is possible, but potentially very difficult.

The compiler includes a macro-language preprocessor that is used without modification. The compiler produces assembler code, but does not include an assembler. A working version of GAS, the GNU Assembler, was not available in time to support CLOCS research.

8.1.2 Assembler

The CLOCS Assembler (CASM) is a combination Flex and C language program. Flex is a regular-expression language translator compatible with the UNIX lex program. It was written by Vern Paxson and contributed to the University of California at Berkeley. Flex extends the capability of lex, it translates programs faster, and the resulting programs run faster. CASM contains about 400 lines of regular expression rules and about 1150 lines of supporting C code and header files.

The assembler takes input from all of the specified files and produces a CLOCS load module. The CLOCS assembler is limited in that all symbols are global. GCC reuses data and branch address symbols, so the assembler cannot handle more than one GCC-produced assembler-source file. In order to avoid writing a linker, and still compile subroutines separately using GCC, an extra processing step was added to make the GCC-produced labels unique to each assembler file.

8.1.3 Implementation-Level Simulator

The CLOCS implementation-level simulator (CAS) is a C language program with 2000 lines of code and 800 lines of supporting header files. The program consists of a top-level main program and several subroutines.

The main program calls memory and memory-management-unit initialization routines, which in turn call a program to load the object module to be simulated. The main program then loops, calling modules that simulate each of the functional units of the implementation, a subroutine to simulate the MMU, and a statistics-gathering routine. After the simulated program terminates or encounters an error condition,

the main program calls a statistics-printing routine and then exits.

A separate subroutine simulates each of the functional units shown in Figure 7.2. Static variables in the main routine simulate communication between the functional units. The order in which the main program evaluates the variables and calls the subroutines simulates the way in which asynchronous signals propagate through the functional units within one machine cycle.

At the beginning of a cycle, if the instruction register is newly updated from memory, modules simulating direct-addressing for operands one and two are activated. The direct addressing modules will set a flag if indirect addressing is required. If indirect addressing is required and the operand register has been updated, then modules simulating indirect addressing are activated. When the operands are ready, a module simulating the arithmetic-logic unit (ALU) is activated.

When CAS runs, several parameters govern the simulation. The MMU uses some of these parameters to set the number of commands that may be executed each cycle (equivalent to the number of memory-system ports), and the latency for read and write operations. CAS also has a debug level that can be set to select: no instruction tracing (just end-of-run statistics), minimal instruction tracing, complete instruction tracing, or maximum information (only useful for debugging the simulator).

The Sun 3/60 is efficient at running these simulations. A simulation of 5000 cycles without producing trace output only takes about 5 seconds.

8.1.4 Scaffolding

The simulation system includes several small programs to accomplish mundane tasks. I refer to these programs as scaffolding because they are useful for getting the job done, but have little use after everything is put together.

CLOCS Module-Dumping Routine

The CLOCS Module-Dumping Routine, CDUMP, loads a CLOCS program into memory, then displays the symbol table, data area, and a disassembled version of the program text. Initially this program was used for debugging the assembler, but it proved most useful for producing the symbol table to analyze simulation traces.

Address Generation

To aid in debugging from simulation traces, a seven-line AWK program reads CLOCS assembler code and writes it out with a hexadecimal address printed at the beginning of the line. This can easily be done by the assembler, but there is no other reason for the assembler to produce listings, and the AWK program is faster to write and debug.

Symbol Reference Debugger

A similar debugging aid is post hoc symbol identification in the simulation traces. A two-line AWK program takes the symbol-table dump produced by CDUMP and generates a new AWK program to identify and print symbols in simulation traces. This works best when the output of the simulator is piped directly to the label-adding AWK program, which then writes it to disk for later analysis. This procedure greatly slows down simulation, extending the time to simulate 5000 cycles to about two minutes. However, two minutes is an acceptable run time for a simulation (hardly time to get a cup of coffee).

8.1.5 Library Routines

The Dhrystone benchmark program expects several standard library routines to be available. Some programs are written in C and compiled to CLOCS assembler language, then hand optimized; others were directly written in CLOCS assembler language. The C and CLOCS assembler source language programs are listed in Appendix B.

Important to the Dhrystone performance were the subroutines "strcpy" and "strcmp." These routines copy and compare character strings, respectively. They optimize well to a few lines of assembler.

The routine "malloc" allocates dynamic storage. The CAS simulator initializes a fixed storage location (default operand segment, offset FFFE00) with a pointer to an area of free storage, and the "malloc" subroutine that was written in CLOCS assembler language maintains the pointer and returns the appropriate value to the calling routine.

The routines "times," "printf," and "scanf" are null or very simple routines. Their functions are not required to obtain results.

8.2 Benchmark Programs

8.2.1 Short Test Programs

A group of very short test programs proved to be useful during simulation-system development, so their performance is reported in Appendix A. “Assign.c” is a simple assignment of a 32-bit integer. “Loop.c” is a simple loop executed 50 times. “Sub.c” is a single subroutine call. These programs contain common constructs, and they are useful to evaluate memory-system performance.

8.2.2 Dhrystone 2.1

The Dhrystone benchmark is a synthetic benchmark program that measures integer performance of computer systems. The number of times that the measured portion of the benchmark program may be executed per second is the *Dhrystones* the architecture may produce. First published in 1984[39], it has been revised by the author to prevent unfair optimizations. For this study, I used version 2.1, dated May 25, 1988. This version consists of two files of C code (dhry_1.c and dhry_2.c) and a header file dhry.h.

Although I do not modify those files, I do modify the behavior of the program by writing specialized library functions for “scanf.” My “scanf” routine always returns a value of one. Thus the program always runs through the timed loop only one time. The library routines “times” and “printf” do not do anything either. I measure the performance of CLOCS using the Dhrystone benchmark program by measuring the number of cycles between calls to “times,” then dividing that number into the estimated clock rate for CLOCS of 16 megacycles per second.

8.3 Simulation Results

Detailed simulation results are presented in the tables in Appendix A. Usually between 2.7 and 3.0 memory references per instruction are required.

8.3.1 Instruction Mix

As shown in Table 8.1, the Dhrystone program uses only 10 of the available 24 instructions. The unused instructions are mostly floating-point and system-control

Operations	Count	Percentage
ADD	489	27.27
SUB	183	10.21
MUL	4	0.21
DIV	1	0.06
OR	1	0.06
LEFT	11	0.61
RIGHT	2	0.11
RIGHTA	12	0.67
MOVE	787	43.89
B	303	16.90
Total	1793	100.00

The Dhrystone program did not use the following operations:
 REM, AND, XOR, ROTATE, TRP, LOB,
 FADD, FSUB, FMUL, FDIV, FIX, FLOAT, or SIZE.

Table 8.1: Percentage of Operations for Dhrystone Benchmark Program

instructions. The unused arithmetic or logical instructions are not commonly used operations in any event. The high percentage of move instructions is surprising. Most of these moves are associated with subroutine calls. The called program is moving the contents of some commonly used addresses to the stack so it may use them as temporary storage locations. Using these locations as registers incurs some procedure-call overhead, but greatly reduces address computation and indirect addressing.

The performance of CLOCS benefited from the short-circuiting feature in the implementation. By using the value remaining in the ALU output register instead of fetching it from memory, the program saved 13% of memory operations. Table 8.2 provides specifics on Dhrystone memory operations. Some programs ("loop.c" and "quicksort.c") realized savings of 30% with short-circuiting.

With the best appropriate compiler optimizations, the Dhrystone benchmark program required 768 instructions in the timed portion of the program. With a simple memory system, those instructions completed in 2220 cycles. At 16Mhz, 2220 cycles per Dhrystone is 7207 Dhrystones/sec.

8.3.2 Effect of Memory System Design

The 7207 Dhrystones/sec quoted above is for a one-port memory system with no read or write delay. This is the simplest system, but an expensive one to build.

Value	Count	Percent
Instruction Fetchs	1793	30.83
Data Fetchs	2533	43.55
Data Stores	1490	25.62
Total Memory	5816	100.00
Actual Memory	5045	86.74
Short Circuits	771	13.26

Table 8.2: Effect of Short Circuits on Memory Access

1 Port Memory				
Write Delay	0	1	2	3
Read Delay 0	100.00	92.93	75.82	62.86
Read Delay 1	73.06	69.22	65.75	56.70
Read Delay 2	49.17	49.17	47.40	45.75
Read Delay 3	37.05	37.05	37.05	36.04
2 Port Memory				
Write Delay	0	1	2	3
Read Delay 0	150.24	134.83	100.32	77.40
Read Delay 1	75.15	75.15	71.09	67.44
Read Delay 2	50.11	50.11	50.11	48.27
Read Delay 3	37.58	37.58	37.58	37.58

Table 8.3: Memory-Design Effect on Dhrystone Performance

Table 8.3 shows the relative performance of CLOCS on the Dhrystone program for many configurations of memory systems.

Because read operations (data or instruction fetches) far outnumber writes, read-operation speed has a greater effect on performance than write speed. In single-ported, multiple-cycle-delay memory systems, writes only have to be one cycle faster than reads to be totally hidden from the system performance. This is because data is ready to be written at the same time that instruction fetch begins. As a result, the write is always delayed one cycle, and if the write takes one fewer cycle than the instruction fetch, it will complete at the same time as the instruction fetch. Only after the instruction is in the instruction registers will further memory operations for operand fetch begin.

In the two-port memory systems, the write will not impact performance as long as it completes in the same time as the read. In this case the writes are hidden behind the

next instruction fetch, so no improvement is observed by making write delay less than read delay. This observation is disappointing, because it shows that posted writes do not help the performance of a CLOCS two-ported memory system. As discussed in Chapter 7, the technique of posted writes is an inexpensive implementation technique to improve performance. It also has the advantage of being reliable. The performance advantage is not affected by bank-interleaving alignment or other hazards of dual-ported memory systems. Also, posted write does not require double performance from the MMU.

The two-ported memory-system performance is also disappointing for the delayed cases when compared to one-ported systems. For one cycle delays, the more complex memory design yields only a 6% difference in performance. Using posted writes with the simpler design reduced this difference to just 2%.

Consequently, the best tradeoff of price-performance is the one-port, one-cycle delay for read and a posted (zero-cycle delay) write. That design achieves a cycles-per-instruction ratio of 3.90 and runs 5342 Dhrystones per second at 16Mhz.

8.3.3 Effect of GCC Optimizations

Table A.5 shows the relative performance of different GCC optimizations. Weicker states that using inline functions is not an appropriate optimization for Dhrystones, so the performance listed above is for "-O -fcombine-regs" only. Using the "-O" option improves performance by 45%, but adding "-fcombine-regs" only adds another 7% to performance.

8.4 Findings: When CLOCS Pays

From the results of the simulations, I can estimate the performance of a real CLOCS computer system. I selected a commercial computer system, the DECStation 3100, for comparison. The 3100 is based on the same MIPS R2000 microprocessor used for comparison in Chapter 6 and operates with a clock frequency of 16 MHz. Using the technology available at the same time that the 3100 was introduced in early 1989, a 16 MHz CLOCS machine could be built with the one-wait state, posted-write memory system described above. Relative manufacturing costs of CLOCS and the 3100 are difficult to estimate, but the integrated circuits for CLOCS memory would be more expensive than the main memory chips used in the 3100. On the other hand, CLOCS

is a simpler design and does not have cache, so the total component count is lower. I estimate that manufacturing costs for CLOCS would be higher, but not more than twice those of the DECStation 3100.

A DECStation 3100 attains approximately 18,000 Dhrystones/sec (slight variations are due to the compiler used and optimizations available). Therefore, the DECStation is $18000/5342 = 3.4$ times faster than a CLOCS. Using techniques we developed[8, 17], we measure the context-switch time of 250 microseconds for a DECStation 3100. Based on estimates from Gallmeister's research[17], CLOCS will switch context in 100 CLOCS instructions, which will take $100 * 3.90/16 = 25$ microseconds.

Now if a task takes X microseconds to run on a DECStation 3100, then it will take $3.4 * X$ to run on a CLOCS. The task will be activated and run in the same time on a CLOCS or a DECStation 3100 if

$$3.4 * X + 25 = X + 250$$

or if X is 95 microseconds. An application must run in less than 95 microseconds after a context switch for it to perform better on CLOCS. In those 95 microseconds, a DECStation 3100 will execute about 1300 machine instructions. This corresponds to a context-switch rate on the DECStation of $1,000,000/(95 + 250) = 2898$. In other words, applications that require context switches more often than 3000 times per second can benefit from the CLOCS architecture.

Consider the example of an ethernet RPC transaction. Our studies[8] indicate that 30% of network overhead is context-switch time. For each context switch, the application needs more than 500 microseconds of CPU processing time on a Sun 4, a computer with approximately the same throughput performance as the DECStation 3100. In this case, the DECStation 3100 can switch context and do the processing in $250 + 500 = 750$ microseconds, and a CLOCS would require $3.4 * 500 + 25 = 1700$ microseconds.

However, if servicing an interrupt takes only 40 instructions, the calculation time is three microseconds. The DECStation requires 253 microseconds and CLOCS requires only $3 * 3.4 + 25 = 35$ microseconds. In this case, the CLOCS architecture provides much better performance.

Figure 8.1 graphically represents this tradeoff. As with Figure 1.2, applications to the left of the dashed line, that is, applications that run in less than 95 microseconds on a DECStation 3100, are activated and run faster on CLOCS.

Thus, from the results of CLOCS throughput performance on simulated bench-

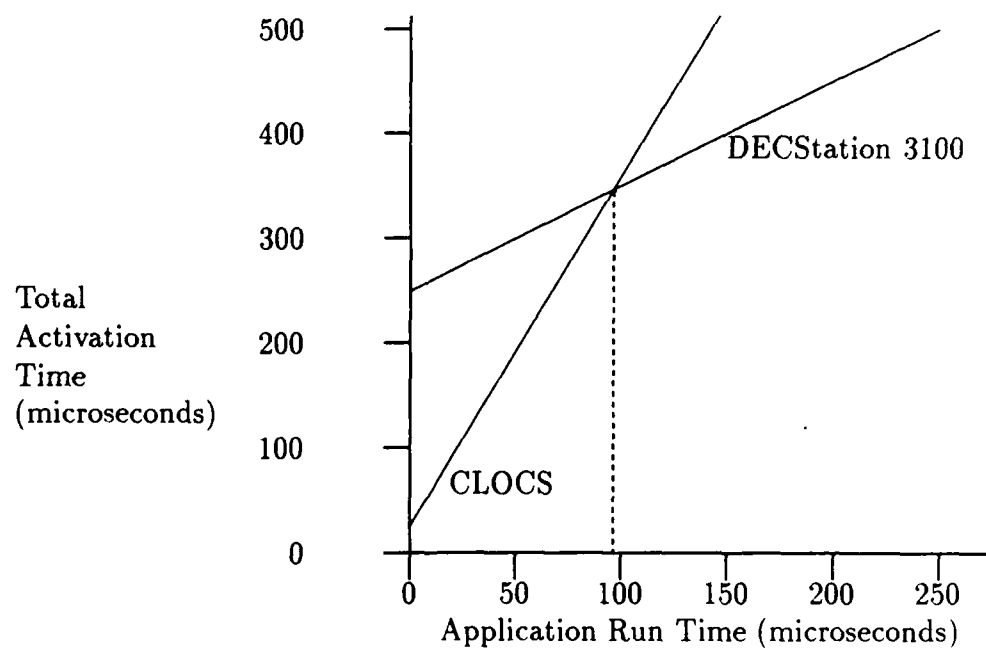


Figure 8.1: Comparison of CLOCS and a DECStation 3100

marks, given an application and the processing time it requires for each activation of a process, we can determine if the application benefits from the CLOCS architecture.

Chapter IX

Conclusions and Future Work

This study revealed much about the issues of context switching. A candidate architecture meeting the main design goals was designed and simulated. Quantitative analyses showed that the architecture has good potential performance. Simulation of an implementation produced estimates of the performance of a feasible CLOCS computer system relative to a commercial workstation. Armed with simulation data, it is possible to evaluate if a given application would run faster on CLOCS than on a computer of conventional design. The study also revealed other observations about context switching and fewer levels in computer memory hierarchy. These observations identified future work.

9.1 CLOCS Potential Performance Is Close to Contemporary RISC

The comparison of CLOCS to a R2000 in Chapter 6 showed that by using assumptions moderately favorable to CLOCS, the new architecture performed 85% as well as the R2000.

9.2 CLOCS Fuller-Type Analysis Score Is High

The score for the CLOCS architecture (using Fuller's quantitative criteria) was much higher than the architectures evaluated by the original study. This indicates that CLOCS has high potential for real-time applications similar to those run on military computers.

9.3 CLOCS Uses 3.2 Memory References per Instruction

Simulation results indicate that programs will average 3.2 memory references per instruction (including fetch of the instruction). This is better than initial bandwidth estimates and indicates potential performance better than 50% of conventional architectures.

9.4 Short-Circuiting Saves 15% of Memory References

The implementation technique of short-circuiting, using the result of one instruction as an operand and avoiding an unnecessary fetch of the data from memory, saves 15% of the memory references. This savings is realized in most reasonable-cases, but could not be considered for worst case analysis.

9.5 Feasible CLOCS System Performance Is Good

The implementation of CLOCS that I designed, supported by the GNU C compiler and other software, performs about 30% as fast as a DECStation 3100. This CLOCS design is conservative, and better performance is possible.

9.6 When Is a New Architecture Indicated?

As discussed in Chapter 8, it is possible to estimate from these results when such an architecture will provide better performance than conventional RISC architectures.

The inequality to verify is:

$$\begin{array}{c} \text{CLOCS Application Time} + \text{CLOCS Context-Switch Time} \\ \text{Less Than} \\ \text{Conventional Application Time} + \text{Conventional Context-Switch Time} \end{array}$$

For the example implementation, CLOCS performs better for applications requiring fewer than 1266 instructions per activation on a DECStation 3100. I expect that comparisons with other contemporary computer systems, such as the Sun 4, would

have produced substantially the same results. This proves the original thesis that a flat memory hierarchy provides better performance for applications that switch context often. This study conservatively defines the nature of applications for which this design approach is superior.

9.7 When a Conventional Architecture Is Indicated

Because conventional architectures perform better when more than approximately 1000 machine instructions are executed per task activation, many common applications do not perform better on CLOCS. Most general purpose time-sharing, communications, and file-serving functions run better on conventional machines. Therefore, a specialized architecture like CLOCS does not make sense for these applications.

Even with small work per activation and many context switches per second, more conventional architectures may still perform better than CLOCS if only a few tasks are activated repeatedly. If the application is known well enough in advance for the designer to estimate the number of desired active processes, it is possible to design a hierarchical memory system that can run more efficiently than CLOCS. For example, the six-legged walking machine[35] requires 66 processes. To support this application a computer only has to handle that many processes to run efficiently. Three designs that can take advantage of known context-switch requirements are shared register files, a special register backing store, and trickle register refill.

9.8 Register File Sharing

If the number of tasks is very small (eight or fewer), then the sharing of the register file by the tasks as is done on the Berkeley RISC and AMD 29000 will perform well. In this approach, the register file is divided into groups of registers, and each group may contain the information for one task. Then each task is assigned one group of registers when it is activated. If the register file is divided into eight groups, eight different tasks can be activated very rapidly. For this approach to be effective, the memory system must also support the same number of active tasks, but since the number of tasks is small, no novel techniques are required. For example, the Sun 4 MMU stores context for 16 processes and would be quite suitable for use with register file sharing.

Registers and the data paths to them consume significant area on integrated

circuits[29]. For larger numbers of processes, more sets of registers would be required and there would not be room to put all of them on a single chip. To keep the state for more processes on a single chip a more compact type of memory could be used as is described next.

9.8.1 Register Backing Store

For larger numbers of tasks, a special backing store for the registers can be added. An example of this is the Intel 80960CA, which implements register windows by having a backing store on the chip connected to the actual register set by two 128-bit wide busses. When a subroutine call is made, these busses are used to save and restore simultaneously sixteen 32-bit registers in only four cycles. During each cycle, four words move from the registers to the backing store, and four move from the backing store to the registers. In this implementation, the backing store contains 16 sets of registers.

Similar techniques could be used to design a machine for fast context switching. A machine with only 16 registers could have a backing store of 192 sets of registers: microprocessors introduced in 1989 commonly have an on-chip cache of that size. As long as fewer than 192 active processes are expected, this design could switch context only slightly more slowly than CLOCS, but throughput would be much higher.

As with register file sharing, the memory system for a machine with fast context switching would have to support all of the active processes. When a process is activated, it would have to use instructions and data in memory, and no extra delay could be added as a result of switching context. With 192 processes, a bigger, more capable MMU than the Sun 4 design would be required. To support virtual memory efficiently, an MMU design like the CLOCS design would be required.

The register backing-store design limits the number of processes based on the area of one integrated circuit. For an unlimited number of active processes, program state must be moved off of the chip. However, fast context-switch performance may be obtained by not reloading all program state at once, as I describe next.

9.8.2 Register Trickle Refill

With trickle register refill the CPU keeps track of the registers that have not been restored every time a context switch occurs. Only when the new program uses a register are its contents fetched from memory. The old value (from the pre-switch

program) could be saved by extra hardware when memory bandwidth is available. As a result, the context switch is very rapid, and a potentially smaller degradation occurs because values are saved and fetched only when required.

High bandwidth to memory is not required, but there are two disadvantages to this scheme. Much complexity must be added to the CPU to keep track of register status and to interlock affected operations. Another disadvantage is that performance is not predictable. A program will run much slower if it is interrupted frequently. Even the first few instructions after activation will run at different rates depending on the status of the register file during execution of the previous program.

It is difficult to predict the context-switch time of this design, because the delay from context switching is spread out over several instructions, and the number of registers referenced between context switches will determine the average context-switch time. Obviously, the average context-switch time depends on the switching rate and the nature of register references for each application. In any event, delay associated with a context switch will always be greater for a register refill design than for CLOCS. In the worst case (every register reference causes a refill) it would also require twice as many data references as CLOCS (save old and fetch new data), so its throughput performance would be almost twice as slow as CLOCS.

The register trickle-refill scheme supports any number of tasks, but, as with register backing store, the MMU must also handle larger numbers of context. It is likely that a CLOCS-style MMU and memory system would be necessary.

Each of the above approaches requires a detailed knowledge of the application set. For register-file sharing and register backing-store designs, a computer designed to run N applications may not be able to handle $N * 2$ applications, even if the context-switch rate remains the same. The register trickle-refill scheme provides lower context-switch performance.

9.9 Programming Language Observations

During compiler porting and simulation runs, I observed two features of the C language that did not work well with the CLOCS architecture: recursion and dynamic memory allocation. Both of these features are used in many but not all languages.

9.9.1 Dynamic Allocation Inefficient

Since the C language assigns variables to dynamic storage by default, CLOCS must use indirect addressing for most of the variables. This reduces the efficiency of the architecture. The estimates made in Chapter 6 assumed that indirect addressing could be avoided in many cases. However, since addresses must be determined at run time, the compiler generates instructions at the beginning of each procedure to calculate the address of each variable. Then, as the subroutine does its computation, it must use these calculated addresses with an indirect-addressing mode to access the variables.

9.9.2 Recursion Works Poorly

Recursion is very inefficient on CLOCS because all of the scratch space for intermediate values and the space necessary for dynamic storage allocation must be in fixed-storage locations. These locations must be copied to the stack for each subroutine call to prevent a corruption of the data during recursive calls.

From examination of the CLOCS assembler-language source produced by the GCC compiler, I estimate that the supporting dynamic variables and recursion added 50% more instructions to the instructions that did the computation.

9.10 Future Work

The interaction of the architecture with programming-language features had a greater effect than expected. The experience obtained in this study revealed several inviting new avenues of research. Although expected improvements would still leave CLOCS with less throughput performance than that of a conventional architecture, any improvement would widen the range of appropriate applications. Three different approaches could improve CLOCS performance by as much as 50%: the existing C language compiler could be improved, the architecture could be improved to run C language programs, or another language could be used that is better suited to the CLOCS architecture.

9.11 Improve the C Language Compiler

Although the GCC compiler does many optimizations, it does not make any attempt to avoid register saving and restoration during subroutine calls. Instead, it relies on machine-dependent routines to implement *callee saves* during subroutine calls. Also, GCC does not allocate registers between subroutines. Instead it uses the same registers in all routines. These features affect CLOCS because the register-allocation routine manages fixed storage-locations used for addresses and scratch space. By enhancing the register allocation function of the C language compiler, these fixed locations could be allocated between subroutines. If only one subroutine uses a storage location it need not be saved unless the procedure is called recursively. This leads to three possible optimizations:

1. The most complete but difficult optimization is to trace the program call tree to determine if a subroutine may be in a recursive chain, and to insert save and restore instructions only in case recursion is possible.
2. One case of non-recursive behavior is easy to identify: if the subroutine is a leaf (does not call other subroutines). In that case, saving of locally used values is never required.
3. Another possible approach is to save the status as to whether a recursive call is in progress. If the program determines that the current call is part of a recursive chain of calls, it may save the fixed locations; otherwise no saving is required.

Determining possible recursion is difficult because the compiler does not necessarily process all subroutines at one time. If two subroutines in separate source files call each other, recursion may be possible, but may undetectable to the compiler since it processes only one file at a time. Also, the compiler must consider more details than just the location of subroutine calls, because the logic of the subroutines may prevent recursion. Thus the first optimization is difficult and potentially too conservative.

The second and third optimizations would be easier to implement using GCC, and the combination of the two would likely provide the same performance improvement as the combination of the first and the third optimizations. I estimate that these and other optimizations discovered by further research could provide a 40% improvement in performance.

9.11.1 Improve the CLOCS Architecture for the C Language

The dynamic storage-allocation feature of the C language is not well supported by CLOCS. Additional addressing modes might make CLOCS much more efficient. Some efficiency could be obtained by adding addressing modes that are stack-pointer or frame-pointer based. This addition would eliminate many of the calculations at subroutine activation. I did not put these into the architecture because of propriety considerations, but their incorporation might improve performance up to 50% by eliminating the instructions added to handle dynamic variables and recursion. Address calculations for dynamic variables could be avoided by placing the variables in a space easily addressed by the new modes. Also, all register save and restore overhead would be removed because the fixed-storage locations for address calculation would no longer be required. These modifications would have to be carefully evaluated; the savings during subroutine initialization may be offset by the larger number of memory references during subroutine execution.

9.11.2 Try Another Language: FORTRAN

It may be that the best performance may be obtained simply by using a language that does not use recursion or dynamic memory. Although such a language is more difficult to use for operating-system work, most real-time applications could easily use one.

If any combination of these improvements resulted in a 50% improvement in CLOCS performance, the crossover point would move by a factor of two to 190 microseconds, as shown by Figure 9.1.

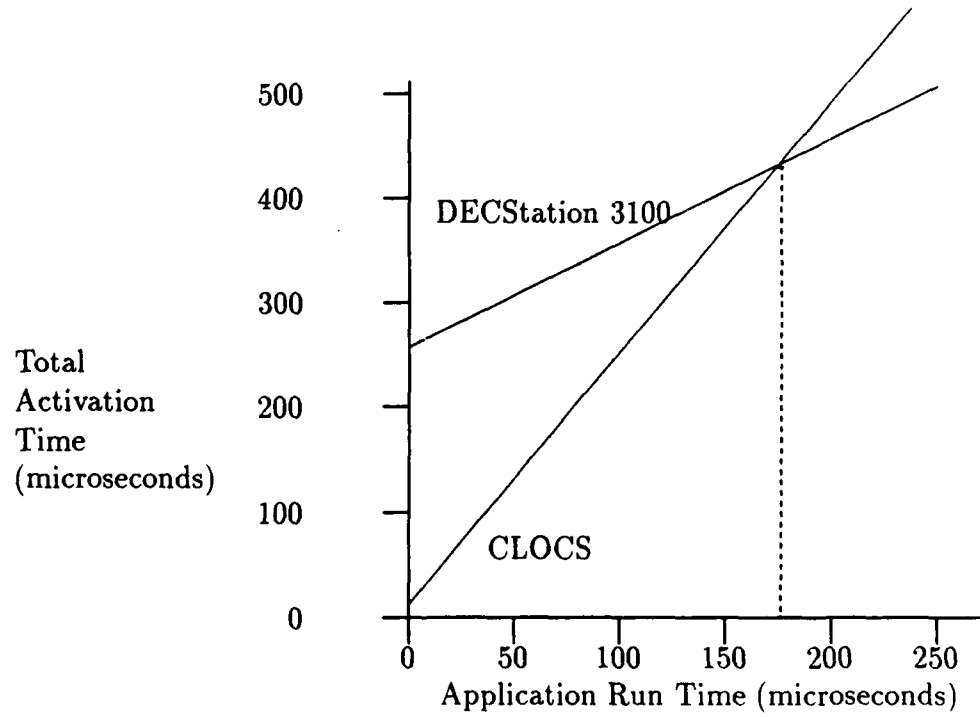


Figure 9.1: Comparison of Improved CLOCS and a DECStation 3100

BIBLIOGRAPHY

- [1] C. G. Bell and W. D. Strecker. Computer Structures: What Have We Learned From the PDP11? In *Proceedings of the Third Annual Symposium on Computer Architecture*, pages 1-14, January 1976.
- [2] Gerrit A. Blaauw. *Digital System Implementation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
- [3] Gerrit A. Blaauw and Frederick P. Brooks Jr. *Computer Architecture*. Addison-Wesley, Reading, Massachusetts, in preparation.
- [4] A. W. Burks, H. H. Goldstine, and J. von Neumann. Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. *Report to U. S. Army Ordnance Department*, 1984.
- [5] G. J. Chaitin. Register Allocation and Spilling via Coloring. Technical Report Research Report RC-9124, IBM Thomas J. Watson Research Center, Yorktown, NY, 1981.
- [6] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation via Coloring. *Computer Languages (British)*, 6:1221-1246, 1981.
- [7] Paul Chow and Mark Horowitz. Architectural Tradeoffs in the Design of MIPS-X. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 300-308, June 1987.
- [8] Mark Davis and Bill O. Gallmeister. The Penalty of Context-Switching in Distributed Computing. Technical Report TR88-025, University of North Carolina, Chapel Hill, May 1988.
- [9] Harvey M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley, Reading, Massachusetts, 1984.
- [10] David R. Ditzel and Alan D. Berenbaum. The Hardware Architecture of the CRISP Microprocessor. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 309-319, June 1987.
- [11] David R. Ditzel and H. R. McLellan. Register Allocation for Free: The C Machine Stack Cache. In *Proceedings of the Symposium for Architectural Support for Programming Languages and Operating Systems*, pages 48-56, March 1982.

- [12] Richard J. Eickemeyer and Janak H. Patel. Performance Evaluation of Multiple Register Sets. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 264–271, June 1987.
- [13] T. Kilburn et al. One-Level Storage System. *IEEE Transactions Electronic Computers*, EC-11(2):223–235, 1962.
- [14] Jerome Feder. The Evolution of UNIX System Performance. *AT&T Bell Laboratories Technical Journal*, 63(8):1791–1814, October 1984.
- [15] Fujitsu Microelectronics, Inc. *MB86900 RISC Processor Architecture Manual*. Fujitsu Microelectronics, Santa Clara, California, 1987. This is really the Sun SPARC Architecture manual.
- [16] Samuel H. Fuller and William E. Burr. Measurement and Evaluation of Alternative Computer Architectures. *Computer*, 10(10):24–35, October 1977. A complete description of how they evaluated nine different computer architectures using quantitative and benchmark analysis.
- [17] Bill O. Gallmeister. The CLOCS Operating System Reference Documents. Technical Report TR88-023, University of North Carolina, Chapel Hill, May 1988.
- [18] John Hennessy, Norman Jouppi, Forest Baskett, and John Gill. MIPS: A VLSI Processor Architecture. Technical Report CSL Tech. Report 223, Stanford University, Stanford, CA, June 1983.
- [19] John L. Hennessy. VLSI Processor Architecture. *IEEE Transactions on Computers*, C-33(12):1221–1246, December 1984.
- [20] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, California, 1989.
- [21] Texas Instruments. *The Texas Instruments 9900 Computer Family*. McGraw-Hill, New York, 1976.
- [22] Gerry Kane. *MIPS R2000 RISC Architecture*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.
- [23] Manolis G. H. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI*. The MIT Press, Cambridge, Massachusetts, 1984.
- [24] Ed Keizer. *Amsterdam Compiler Kit Reference Manual*. UniPress Software, Edison, New Jersey, 1987.
- [25] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.
- [26] Earl Killian. Statistics of Timberwolf running on R2000. Personal Correspondence, October 1987.

- [27] Sam Leffler, Mike Karcels, and M. Kirk McKusick. Measuring and Improving the Performance of 4.2BSD. In *Editorial Materials - Number 1, 4.2BSD Internals*. USENIX Technical Conference, Dallas, Texas, January 1984.
- [28] Dennis W. Leinbaugh. Guaranteed Response Times in a Hard-Real-Time Environment. *IEEE Transactions on Software Engineering*, 1980.
- [29] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley, Reading, Massachusetts, 1980.
- [30] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*, 29(3):184-201, March 1986.
- [31] Motorola, Inc. *Technical Summary, 32-Bit Third Generation RISC Microprocessor*. Motorola, Phoenix, Arizona, 1988.
- [32] D. A. Patterson and C. H. Sequin. RISC-I: A Reduced Instruction Set VLSI Computer. In *Proceedings of the Eighth Annual Symposium on Computer Architecture*, pages 443-449, May 1981.
- [33] David A. Patterson and Carlo H. Sequin. A VLSI RISC. *Computer*, 18(9):8-21, September 1982. Later, more complete information on RISC I with more details about the implementation.
- [34] George Radin. The 801 Minicomputer. *IBM Journal of Research and Development*, 27(3):237-247, May 1983. Early work. Contains classic considerations (move functions from hardware to software; evaluate use of all functions, move functions from run time to compile time).
- [35] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregory Taulbee. High-Performance Operating System Primitives for Robotics and Real-Time Control Systems. *ACM Transactions on Computer Systems*, 5(3):189-231, August 1987.
- [36] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. *Computer Structures: Principles and Examples*. McGraw-Hill, New York, 1982.
- [37] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Boston, 1989.
- [38] Shreekanth S. Thakkar and Alan E. Knowles. A High-Performance Memory Management Scheme. *IEEE Computer*, 8(5):8-20, May 1986.
- [39] Reinhold P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013-1030, October 1984. The first paper on Dhrystones.

Appendix A

Detailed Simulation Results

A.1 General Performance

The general performance for Dhrystones, Assign, Lloop, Sub, and Quicksort are shown in Table A.1. These statistics were printed by the CAS simulation program.

A.2 Various Memory Designs

Table A.2 contains the number of cycles for each of the programs to run using different memory systems. The Dhrystone portion of this data is presented in normalized form in Table 8.3. Table A.3 contains the cycles per instruction for each case.

A.3 Effect of GCC Optimizations

The effects of various optimizations provided by GCC are displayed in Table A.4. A more compact presentation is given in Table A.5. Keep in mind that the author of the Dhystone benchmark program states that procedure inlining is not an appropriate optimization when comparing systems. When comparing CLOCS with the DECStation 3100, I use values obtained without use of "-finline-functions."

As expected, the standard GCC optimizations provide the greatest incremental improvement. The combine-regs option looks for the special case of a register copied to another register. This generally adds an incremental improvement of 3% by fixing redundancies introduced by address calculations.

Compiled with gcc -O	dhystone	assign	loop	sub	quicksort
General Counts					
Total Cycles	5046	26	498	85	99999
Total Operations	1793	9	218	30	38003
Cycles Per Instruction	2.8142	2.8889	2.8440	2.8333	2.6313
Total MMU Operations	5046	26	498	85	99999
Total Memory Ops	5046	26	498	85	99999
Operations (Number of Occurrences)					
ADD	489	1	52	5	9441
SUB	183	2	54	6	4785
MUL	4	0	0	0	0
DIV	1	0	0	0	0
OR	1	0	0	0	0
LEFT	11	0	0	0	4693
RIGHT	2	0	0	0	0
RIGHTA	12	0	0	0	4694
MOVE	787	0	59	16	9661
B	303	1	53	3	4728
Operand 1 Address Modes (Number of Occurrences)					
RELATIVE	570	3	107	10	18983
INDIRECT	288	2	3	5	4759
ZERO	83	0	0	1	12
ABS INDIRECT	84	1	1	2	11
IMMEDIATE	768	3	107	12	14238
Operand 2 Address Modes (Number of Occurrences)					
RELATIVE	1340	8	216	25	37881
INDIRECT	453	1	2	5	122
Short Circuits (Number of Occurrences)					
psw to MMU_1	83	0	0	1	12
alu to MMU_1	197	3	54	7	9407
alu to MMU_2	574	1	105	6	23572

Table A.1: General Performance Results of CLOCS

Write Delay	0		1		2		3	
Ports	1	2	1	2	1	2	1	2
Dhrystones								
0 Read Delay	5048	3360	5432	3744	6658	5032	8030	6522
1 Read Delay	6909	6717	7293	6717	7677	7101	8903	7485
2 Read Delay	10266	10074	10266	10074	10650	10074	11034	10458
3 Read Delay	13623	13431	13623	13431	13623	13431	14007	13431
assign								
0 Read Delay	47	32	51	36	65	50	80	65
1 Read Delay	61	61	65	61	69	65	83	69
2 Read Delay	90	90	90	90	94	90	98	94
3 Read Delay	119	119	119	119	119	119	123	119
lloop								
0 Read Delay	933	656	1089	812	1366	1089	1643	1366
1 Read Delay	1309	1309	1465	1309	1621	1465	1898	1621
2 Read Delay	1962	1962	1962	1962	2118	1962	2274	2118
3 Read Delay	2615	2615	2615	2615	2615	2615	2771	2615
sub								
0 Read Delay	128	85	138	95	178	135	220	177
1 Read Delay	168	167	178	167	188	177	228	187
2 Read Delay	250	249	250	249	260	249	270	259
3 Read Delay	332	331	332	331	332	331	342	331
quicksort								
0 Read Delay	32284	20745	36667	25128	43753	33281	52985	42600
1 Read Delay	43686	41487	48069	41487	52452	45870	59538	50253
2 Read Delay	64428	62229	64428	62229	68811	62229	73194	66612
3 Read Delay	85170	82971	85170	82971	85170	82971	89553	82971

Table A.2: Cycle Counts for Various Programs and Memory Systems

1 Port Memory				
Write Delay	0	1	2	3
Read Delay 0	2.90	3.12	3.83	4.62
Read Delay 1	3.97	4.19	4.41	5.12
Read Delay 2	5.90	5.90	6.12	6.35
Read Delay 3	7.83	7.83	7.83	8.05

2 Port Memory				
Write Delay	0	1	2	3
Read Delay 0	1.93	2.15	2.89	3.75
Read Delay 1	3.86	3.86	4.08	4.30
Read Delay 2	5.79	5.79	5.79	6.01
Read Delay 3	7.72	7.72	7.72	7.72

Table A.3: Memory Design Effect on Dhrystone Cycles per Instruction

None				
Memory System	Start	Stop	Diff	Dhrystones
Instruction Count	438	1551	1113	
1 Port, 0 Delay	1304	4620	3316	4825
1 Port Posted	1843	6299	4456	3590
1 Port, 1 Delay	1900	6588	4688	3413
2 Ports, 0 Delay	889	3051	2162	7401
2 Ports, 1 Delay	1777	6101	4324	3700

gcc -O				
Memory System	Start	Stop	Diff	Dhrystones
Instruction Count	423	1211	798	
1 Port, 0 Delay	1266	3552	2286	6999
1 Port Posted	1798	4882	3084	5188
1 Port, 1 Delay	1848	5089	3241	4936
2 Ports, 0 Delay	866	2356	1490	10738
2 Ports, 1 Delay	1731	4711	2980	5369

gcc -O -fcombine-regs				
Memory System	Start	Stop	Diff	Dhrystones
Instruction Count	423	1191	768	
1 Port, 0 Delay	1266	3496	2230	7207
1 Port Posted	1798	4793	2295	5342
1 Port, 1 Delay	1848	5003	3155	5071
2 Ports, 0 Delay	866	2313	1447	11057
2 Ports, 1 Delay	1731	4625	2894	5528

gcc -O -fcombine-regs -finline-functions				
Memory System	Start	Stop	Diff	Dhrystones
Instruction Count	423	1161	738	
1 Port, 0 Delay	1266	3401	2135	7494
1 Port Posted	1798	4683	2885	5545
1 Port, 1 Delay	1848	4884	3036	5270
2 Ports, 0 Delay	866	2260	1394	11477
2 Ports, 1 Delay	1731	4519	2788	5738

Note: -fstrength-reduce had no effect
on the timed portion of Dhrystones.

Table A.4: GCC Optimizations for Dhrystones

Optimizations	1 Port 0 Delay	1 Port Posted Write	1 Port 1 Delay	2 Ports 0 Delay	2 Ports 1 Delay
None	100.00%	100.00%	100.00%	100.00%	100.00%
Add -O	145.05%	144.49%	144.62%	145.10%	145.10%
Add combine-regs	149.37%	148.78%	148.58%	149.41%	149.39%
Add inline-functions	155.31%	154.45%	154.41%	155.08%	155.07%

The results from Table A.4 are shown here normalized to the no optimization cases and expressed as percentages.

Table A.5: Effect of GCC Optimizations (Percentages)

Appendix B

Source Listings

B.1 Arrangement of Support Software

The CLOCS architecture was simulated using several pieces of software. These included:

CCC The C language compiler produced CLOCS assembler code.

GCCC The Gnu C language compiler also produced CLOCS assembler code.

CASM The CLOCS assembler produced a CLOCS object module.

CLOADER The CLOCS loader reads an object module into memory.

CDUMP This simple dumping program reads and disassembles object modules.

CAS The CLOCS implementation-level simulator produces a trace of the simulated execution of object modules.

Each of these programs had to work with the next one; writing the programs for reliable communications was challenging.

B.1.1 CCC (C Language Compiler for CLOCS)

The C language compiler was initially written by three graduate students taking an advanced compiler course. They produced the compiler as their class project. They used the Amsterdam Compiler Kit (ACK)[24] to facilitate the creation of the compiler. Unfortunately, a CLOCS assembler and simulator were not available by the time the project had to be presented to the instructor. A small test program was run through the compiler, hand assembled and run on a very high-level architectural simulator. The tediousness of hand-assembly limited the testing and debugging done by the class-project participants. Several bugs and undesirable features were later

corrected as the assembler began operations and a comprehensive C library was compiled.

As C library functions were compiled and output from benchmark and other test programs was examined, it became clear that ACK would not be a satisfactory tool for this research. Much work remained before it could support multiple data types, and the output was poorly optimized, preventing a good assessment of the capabilities of the CLOCS architecture from the results of benchmark-program performance.

B.1.2 GCCC (GNU C Language Compiler for CLOCS)

The Gnu C compiler[37] was also examined. This high-quality compiler had the advantages of containing more optimizations and being available in C language source form with good documentation. This compiler was also well supported. Although the compiler was in development at the time, it was relatively free of bugs, and only one bug ever impeded progress. Several very helpful people were available locally and via electronic mail to give advice and offer solutions to the more difficult problems of porting this compiler.

B.1.3 CASM (CLOCS Assembler)

CASM, the CLOCS Assembler, was written using lex. The lex program and its C language support routines produce an object module in main memory and then write a CLOCS object module.

B.1.4 CLOADER (CLOCS Loader) and CDUMP (CLOCS Object Module Dumper)

CLOADER is a subroutine that loads a CLOCS object module into main memory. CDUMP produces human readable output of CLOCS object modules. It tested CLOADER and provided data for the debugging of CASM.

B.1.5 CAS

The CLOCS Architectural Simulator (CAS) is a large C language program to simulate implementations of the CLOCS architecture. The program was initially written to simulate the two-instruction pipeline implementation. However, minor modifications

adapted it to the final implementation. The simulator supports several different models of main memory; the number of ports and the delay of reads or writes are set as the simulation begins. CAS uses CLOADER to get the object module loaded, does some MMU initialization and then simulates the program, producing trace results and summaries of instruction use and address modes.

B.2 Arrangement of Application Software

For the purposes of this appendix, application software consists of the programs that run on the CLOCS machine.

B.2.1 C Library Routines

The C language supports many data manipulations and all operating-system services with calls to standard subroutines[25]. The most important data manipulations routines for this research are the character string compare “strcmp” and character string move “strcpy” routines. These routines are called in the timed portion of the Dhrystone benchmark program, so they have to be reasonably efficient. The Dhrystone also uses the operating service call to obtain memory “malloc” because some of the variables it uses are dynamically allocated. The Dhrystone program also calls “scanf” to ask the number times to run the timed loop, “printf” for printing results, and “times” to calculate the elapsed time. I supplied dummy subroutines for the complex functions because I could obtain results without writing complicated subroutines to be called only once.

“strcmp”

Figure B.1 shows the C language source[25] for the “strcmp” routine. After it was compiled by GCC, I hand optimized it to remove unnecessary saving of parameters on the stack. The CLOCS assembler source code is in Figure B.2.

“strcpy”

The routine “strcpy” was generated the same way as “strcmp.” Figure B.3 contains the C language source, and Figure B.4 contains the CLOCS assembler source code.

```

/*
 * STRCMP from K&R version 2
 * page 106
 */

/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}

```

Figure B.1: Strcmp.c Source

“malloc”

The CLOCS simulator establishes several extra pages of virtual memory and stores the location of this data space in a standard location in the user's data segment. The “malloc” subroutine may use this pointer and the corresponding data pages to provide dynamic memory to the running program. By adding this capability to the simulator, I avoided the necessity of writing a program to do this virtual-memory allocation dynamically. Writing such a program is not difficult, but it is time-consuming. The simple “malloc” program source is listed in Figure B.5.

“scanf”

The Dhrystone benchmark program calls the “scanf” routine to get the user's input of the number of iterations of the timed loop. I replaced the standard routine with a C program that just sets the value to one. The source is shown in Figure B.6. I also hand optimized this subroutine into the assembler language program shown in Figure B.7.

“printf” and “times”

The standard C language subroutines “printf” and “times” are called by the Dhrystone Benchmark program. Since data collection does not require these routines to work, null subroutines are substituted. Figures B.8 and B.9 list these programs.

```

; Compiled_By_GCCC.:
.text
.align 8
.globl _strcmp
_strcmp:
; BEGIN NEW FUNCTION.
b STRCMP_L2
STRCMP_L6:
bqine STRCMP_L4,@R_48
movsi <0,RTN
b STRCMP_L1
STRCMP_L4:
adddi <1,R_48
adddi <1,R_49
STRCMP_L2:
movqi @R_48,RTN
subqi @R_49,RTN
bqieq STRCMP_L6,RTN

; Sign extension is a drag.  At least it only has to be done once
leftsi <24,RTN
rghtasi <24,RTN
STRCMP_L1:
movdi @STACK_PTR,BRANCH_TGT
b %@BRANCH_TGT
; END THIS FUNCTION.

```

Figure B.2: Strcmp.s Source

```

/*
 * STRCPY from K&R version 2
 * page 106
 */

/* copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
while(*s++ = *t++)
;
}

```

Figure B.3: Strcpy.c Source

```

; Compiled_By_GCCC.:
; From K&R, version 2, page 106, then hand optimized.
; Changed to jump into middle to lower loop overhead
.text
.align 8
.globl _strcpy
;_strcpy:
; BEGIN NEW FUNCTION.
; b STRCPY_START
STRCPY_L2:
adddi <1,R_49
adddi <1,R_48
_strcpy:
STRCPY_START:
movqi @R_49,@R_48
bqine STRCPY_L2,@R_48
movdi @STACK_PTR,BRANCH_TGT
b %@BRANCH_TGT
; END THIS FUNCTION.

```

Figure B.4: Strcpy.s Source

```

; Hand assemble by Mark C. Davis
; 1/16/90

.text
.align 8
.globl _malloc
_malloc:
; BEGIN NEW FUNCTION.
; Register 48 (first parameter) contains size of desired storage.

; normalize SI to DI for addition
rghtdi <32,R_48
movdi 0XFFFE00,RTN
adddi R_48,0XFFFE00
movdi @STACK_PTR,BRANCH_TGT
b %@BRANCH_TGT
; END THIS FUNCTION.

```

Figure B.5: Malloc.s Source

```

/* A Dummy scanf subroutine to set the drystone to one */
/* iteration. */

/* Mark Davis 1/25/90 */

void scanf( char *fmt, int *n)
{ *n = 1;}

```

Figure B.6: Scanf.c Source

```

; Compiled_By_GCCC.:
; Then Hand Optimized by Mark Davis
; 1/25/90

```

```

.text
.align 8
.globl _scanf
_scanf:
; BEGIN NEW FUNCTION.
movsi <1,@R_49
movdi @STACK_PTR,BRANCH_TGT
b %@BRANCH_TGT
; END THIS FUNCTION.

```

Figure B.7: Scanf.s Source

```

; Hand assemble by Mark C. Davis
; 1/16/90

```

```

.text
.align 8
.globl _printf
_printf:
; BEGIN NEW FUNCTION.
; Don't Do anything except set RTN code to zero
movdi <0,RTN
movdi @STACK_PTR,BRANCH_TGT
b %@BRANCH_TGT
; END THIS FUNCTION.

```

Figure B.8: Printf.s Source


```

; Hand assemble by Mark C. Davis
; 1/16/90

.text
.align 8
.globl _times
_times:
; BEGIN NEW FUNCTION.
; Don't Do anything except set RTN code to zero
movdi <0,RTN
movdi @STACK_PTR,BRANCH_TGT
b %@BRANCH_TGT
; END THIS FUNCTION.

```

Figure B.9: Times.s Source

```

void main() {
long i =1;
}

```

Figure B.10: Assign.c Source

B.2.2 Test Programs

During the writing and debugging of the simulation system, I wrote many small C language programs to test various features. Four of the programs provide interesting simulations results, so I include their performance statistics in Appendix A and their source here. The source for "assign," a simple assignment program, is shown in Figure B.10. Figure B.11 presents the source for a small looping program call "lloop." To test and evaluate performance of subroutine calls I wrote "sub," the program in Figure B.12. The "quicksort" program is a minor modification of a test program used by a group of students who wrote a CLOCS C language compiler for a course. The program is listed in Figures B.13 and B.14. Many other programs were written and run, but are not of sufficient interest to include here.

```

main()
{
    int i,j;

    for (i=0; i < 50; i++) j += 3;

}

```

Figure B.11: Lloop.c Source

```

int sp();

void main() {
    int i=1;
    i=sp(i);
    i+=1;
}

int sp(ip)
int ip;
{
    return ip;
}

```

Figure B.12: Sub.c Source

```

int a[100];

void swap(int x,int y)
{
    int temp;
    temp = a[x];
    a[x] = a[y];
    a[y] = temp;
}

void sort(int bottom,int top)
{
    int low = bottom,high = top;
    int mid = (low + high)/2;
    int pivot = a[mid];
    while (low < high)
    {
        while (a[low] < pivot)
            low = low + 1;
        while (a[high] > pivot)
            high = high - 1;
        if (low < high)
        {
            swap(low,high);
            low++;
            high--;
        }
    }
    if (bottom < top)
    {
        sort(bottom,mid);
        sort(mid+1,top);
    }
}

```

Figure B.13: Quicksort.c Source (Subroutines)

```

main()
{
    int bottom = 0, top;
    int num;
    long long int i;
    num=8;
    a[0]=12;
    a[1]=63;
    a[2]=3;
    a[3]=13;
    a[4]=57;
    a[5]=31;
    a[6]=61;
    a[7]=11;
    sort(bottom,num-1);
    for (i=0;i < num;i++)
        top=a[i];
}

```

Figure B.14: Quicksort.c Source (Main Program)

B.2.3 Dhrystones Version 2.1

I used the Dhrystone source as distributed. However, the null "printf" and "times" library routines described prevent the program from determining and printing the results. Simulator timings are used instead to determine Dhrystone performance. I obtained the source from a Usenet posting by Rick Richardson (return electronic mail address ...!seismo!uunet!pcrat!rick) dated 4 Dec 88. The source is available from PC Research, Inc, at 94 Apple Orchard Drive, Tinton Falls, NJ 07724 and from the server at netlib@mcs.anl.gov.